Woldia University

Institute of Technology (IoT)

Department of Computer Science

Programming Chair

Computer Programming in C++

By

Derso M. (MSc in SE)

May 21, 2021

Chapter One

Introduction to Computer Programming

Language

- **Human Language:**
  - Commonly used to express feelings and understand other person expressions.
  - It can be oral or gestural kind of communication

- **Computer Language:**
  - Computer languages are the languages by which a user command a computer to work on the algorithm which a user has written to get an output.

What is a program, programming, and programmer?

- **Program:**
  - Usually, one or more algorithms written in a programming language that can be translated to run on a real machine.
  - We sometimes call programs software.
  - A **computer program** is a series of organized instructions that directs a computer to perform tasks.

- **Programmer** is someone who writes computer program.
  - Programmers write, test, and maintain programs or software that tell the computer what to do.

- **Programming** is the process of writing computer programs for various purposes.

What skills are required to become a programmer?

- For someone to be a programmer, in addition to basic skills in computer, needs to have the following major skills:

  - **Programming Language Skill:** knowing one or more programming language to talk to the computer and instruct the machine to perform a task.

  - **Problem Solving Skill:** skills on how to solve real world problem and represent the solution in understandable format.

  - **Algorithm Development:** skill of coming up with sequence of simple and human understandable set of instructions showing the step of solving the problem.

Programming language

- A programming language is a set of words, symbols and codes that enables humans to communicate with computers.
- Hundreds of programming languages exist today. Each language has its own standard or rules for writing the commands and/or instructions.
- A programming language is some what like a natural language, but with a very limited set of statements and strict syntax rules.
- It has statements to implement sequential, conditional and iterative processing - **algorithms**.
- Examples: FORTRAN, COBOL, Lisp, Basic, Pascal, C, **C++**, Java, C#, Python, …

Programming language

- You eventually need to convert your program into machine language so that the computer can understand it.
- There are two ways to do this:
  - Compile the program
  - Interpret the program

- **Compile** is to transform a program written in a high-level programming language from source code in to object code.
- This can be done by using a tool called **compiler**.
- A compiler reads the whole source code and translates it into a complete machine code program to perform the required tasks which is output as a new file.

Programming language

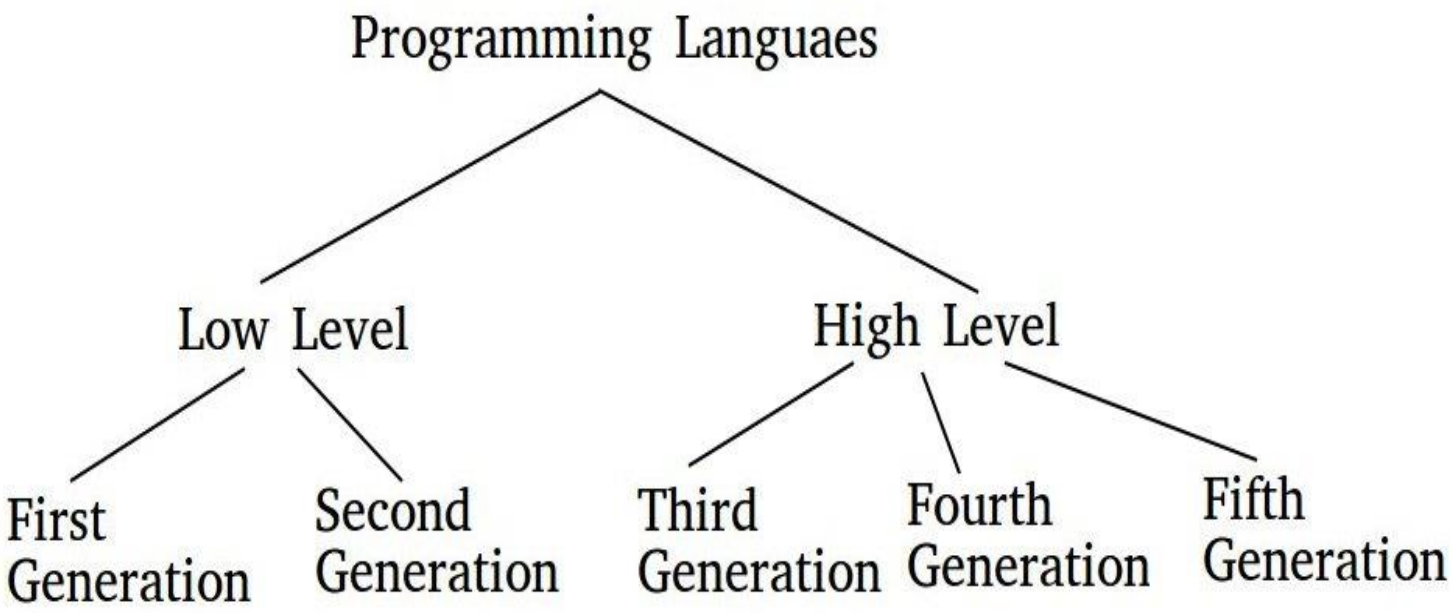
- **Interpreter** is a program that executes instructions written in a high-level language.

- An interpreter reads the source code one instruction or line at a time, converts this line into machine code and executes it.

Generation of programming languages

- There are five generation of Programming languages. They are:
- **First Generation Languages:**
  - These are low-level languages like machine language.
- **Second Generation Languages:**
  - These are low-level assembly languages used in kernels and hardware drives.
- **Third Generation Languages:**
  - These are high-level languages like C, C++, Java, Visual Basic and JavaScript.
- **Fourth Generation Languages:**
  - These are languages that consist of statements that are similar to statements in the human language. These are used mainly in database programming and scripting. Example of these languages include Perl, Python, Ruby, SQL, MatLab (Matrix Laboratory).
- **Fifth Generation Languages:**
  - These are the programming languages that have visual tools to develop a program. Examples of fifth generation language include Mercury, OPS5, and Prolog.

Generation of programming languages



Assignment I: ***Detail description of the generation of programming language.***

Problem solving using computers

- **Problems:** Undesirable situations that prevent an organization from fully achieving its purpose, goals and objectives

- There are two approaches of problem solving:

  - **Top-down approach:** focuses on breaking down a big problem into smaller and understandable chunks.

  - **Bottom-up approach:** first focuses on solving the smaller problems at the fundamental level and then integrating them into a whole and complete solution.

Programming paradigm

- **Structured/modular Programming:**
  - Divides a program into a set of functions or modules.
  - These functions have statements embraced inside curly braces.
  - Each of these functions performs a subtask.
  - Usually, as each function represents a specific functionality.

- **Object oriented programming:**
  - Allows the programmer to represent real-world scenarios using objects.
  - An object is any entity that has ***states*** and ***behaviours***.
    - **States** represent the attributes or data of an object,
    - **Methods** represent the behaviours of objects.
  - Student, Employee, Book etc. are objects.
  - For example, to create an Employee object, there should be an Employee class.

Programs we design need to be

- **Reliable**: the program should always do what it is expected to do and handle all types of expectations
- **Maintainable**: the program should be in a way that it could be modified and upgraded when the need arises.
- **Portable**: it needs to be possible to adapt the software written for one type of computer to another with minimum modification.
- **Efficient**: the program should be designed to make optimal use of time, space and other resources.

Software development life cycle

- The method (approach) that software Engineers use in solving problems in computer science is called the software development method or software life cycle.
- The software life cycle has the following components
  - **Preliminary Investigation** (Defining the problem, feasibility study) - technical, economical, operational, etc.
  - **Analysis (Requirement gathering) -** try to understand the business in general (activities done, how it is done, etc.)
  - **Design -** develop a series of steps with a logical order
  - **Implementation -** translate (code) each step of the algorithm
  - **Testing -** Will the system produce the desired results?
  - **Maintenance -** fix bugs found by the customer, make modifications, add new features
  - **Documentation -** written descriptions of, specifications, development, and actual code of program.

Algorithm development

- An **algorithm** is procedure for solving a problem in terms of the action to execute (what to do) and the order in which these actions are executed(done)
- An algorithm needs to be
  - **Precise** and **unambiguous** (no ambiguity in any instruction and in the order of execution)
  - **Simple**
  - **Correct**
  - **Finite** (has to have an end)
  - **Produce expected output**
  - **Efficient**: in time, memory and other resources
- An algorithm can be expressed in many ways. Some of these methods are narrative, flowchart and pseudo-code.

Algorithm development

- **Narrative**: can be understood by any user who may not have any knowledge of computer programming.
- **Flowcharts**: a diagram consisting of labelled symbols, together with arrows connecting one symbols to another.

Terminal point marks the beginning or end of a program

Process: is used to compute/calculate a process.

Decision: Indicates a yes/no decision to be made by the program

Input/output: is used to show input or output data

Flow line: is used to show the direction of logical flow

On-page Connector: used to connect two points without drawing a flow line

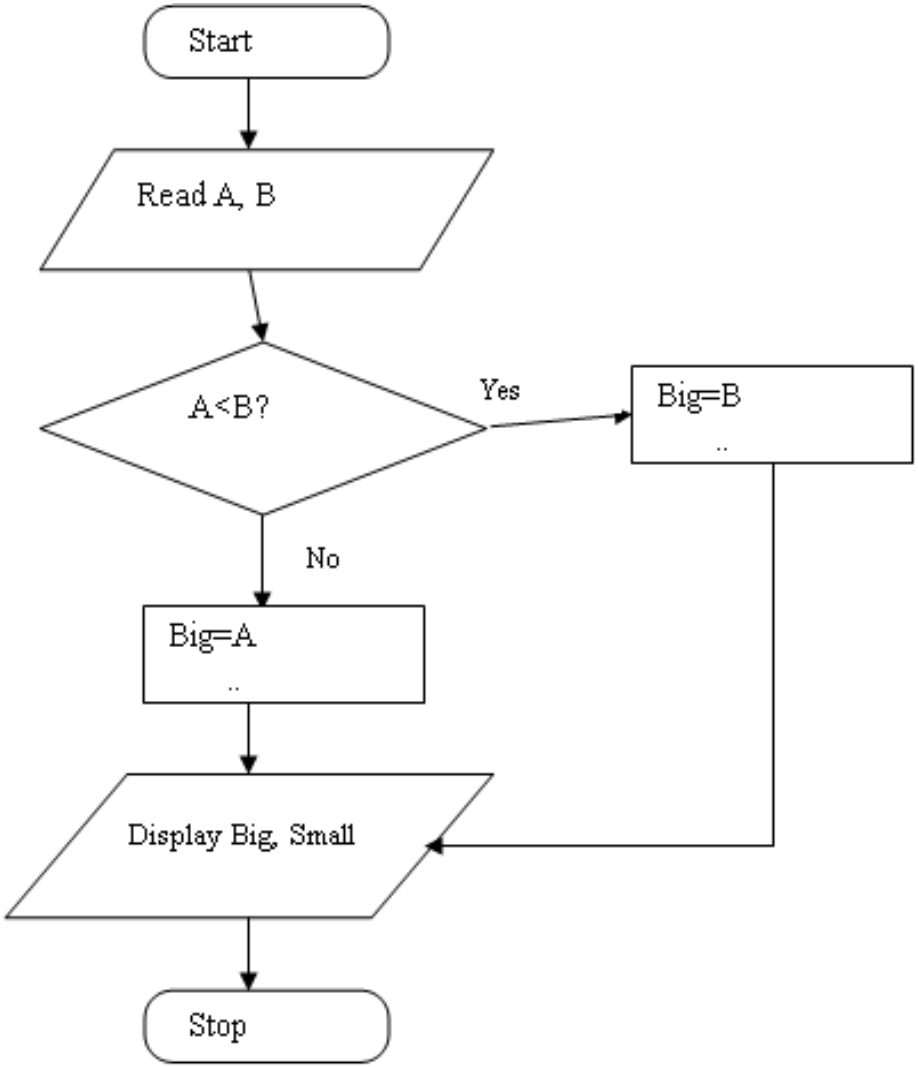
Annotation flag: used to add clarifying comments or descriptions

Inter-page Connector: used as exit or entry from a flowchart on one page to a flowchart in another page

Predefined Process: indicates a process defined else where

Algorithm development

- **Example:** a program that identifies a larger and smaller number from two numbers



Algorithm development

- **Pseudo-code:** it is much similar to real code. We use verbs to write pseudo-code. Capitalize important words that show actions.

```
A pseudo-code that calculates grade
        ACCEPT Mark, Name
        IF Mark>80 Then
          Grade←A
        ELSE IF Mark>=70 Then
          Grade←B
        ELSE IF Mark>=60 Then
          Grade←C
        ELSE IF Mark>=50 Then
          Grade←D
        ELSE
          Grade←F
        ENDIF
DISPLAY Grade, Name
```

Chapter Two

C++ Basics

Outlines

- C++ terminology and syntax
- The complete development cycle in C++ or C++ IDE
- A simple C++ program description
- Basic Elements
  - Input/output in C++
  - Keywords
  - Identifiers
  - Variables
  - Basic Data Types
  - Constants
  - Comments
- Operators
- Operator Precedence

C++ Terminology and Syntax

- **Pre-processor** directives are not regular code lines with expressions but indications for the compiler's pre-processor.
  - It is processed before compiling the program and not terminated by semicolon.
- **Statement** is a simple or compound expression that can actually produce some effect. It must be terminated by a semicolon (;)
- **Block** is a group of programming statements enclosed by braces { }.
- **Comments** are not executable statements and are ignored by the compiler; but they provide useful explanation and documentation.

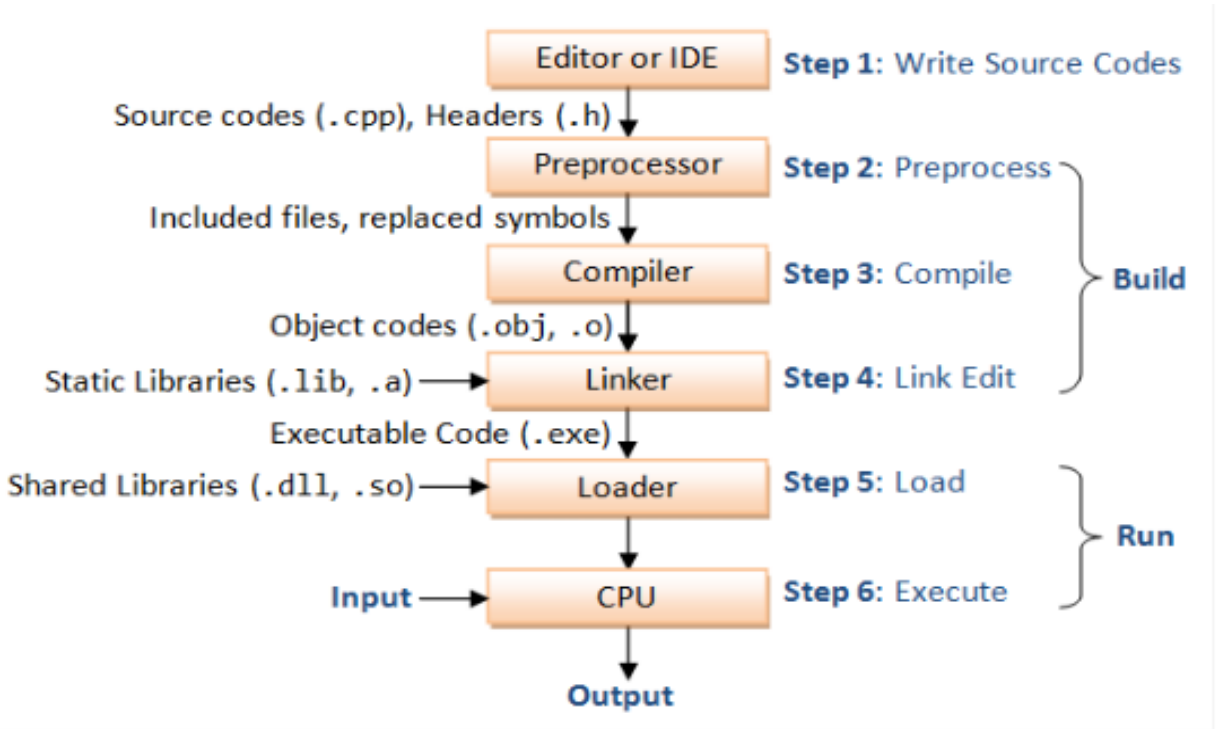
C++ Terminology and Syntax

- **Whitespaces** includes Blank, tab, and newline are collectively called whitespaces.
- C++ is **case sensitive** - a ROSE is NOT a Rose, and is NOT a rose.
- **Syntax** consists of the rules for the correct use of the language.
- **Semantics** are meanings given to syntactically correct constructs of the language.

C++ IDE / The complete development cycle in C++

Step 1: Write the source codes (`.cpp`) and header files (`.h`).

Step 2: Pre-process the source codes according to the *preprocessor directives*. Preprocessor directives begin with a hash sign (#), e.g., `#include` and `#define`. They indicate that certain manipulations (such as including another file or replacement of symbols) are to be performed BEFORE compilation.



Step 3: Compile the pre-processed source codes into object codes (`.obj`, `.o`).

Step 4: Link the compiled object codes with other object codes and the library object codes (`.lib`, `.a`) to produce the executable code (`.exe`).

Step 5: Load the executable code into computer memory.

Step 6: Run the executable code, with the input to produce the desried output.

C++ IDE / The complete development cycle in C++

- **Summary**
  - Start with C++ source code files (***.CPP***, ***.HPP/.H***)
  - Compile: convert source code to object code
    - Object code stored in object file (***.OBJ*** or ***.O***)
  - Link: combine contents of one or more object files (and possibly some Libraries) to produce executable program
  - Executable program can then be run directly

Structure of C++ Program

- **A C++ program has the following structure**

[Comments]

[Pre processor directives]

[Global variable declarations]

[Prototypes of functions]

[Definitions of functions]

```
#include<iostream.h>
int num1; //is global for this program
int add(int x, int y, int i); //function declaration
int add( int x, int y) //function definition
{
        int z; /*z is local variable and accessed only inside this
        add(int x, int y) function */
        ….
}
int main()
{
        float num2; /* num2 is local variable and accessed only
        inside this main() function */
         cout<<"\n Enter your Department:";
         …
return 0;
}
```

Sample C++ Program Description

- *// my first program in C++* is a comment and do not have any effect on the behaviour of the program.
- *#include <iostream>* tells the pre-processor to include the iostream standard file.
  - basic standard input/output library in C++
- *using namespace std;* all the elements of the standard C++ library are declared within what is called a namespace, the namespace with the name std.
- *int main ()* is the beginning of the definition of the main function.
  - other functions with other names defined before or after it.
  - the instructions within always be the first ones to be executed.

```
// my first program in C++

#include <iostream>
using namespace std;

int main ()
{
  cout << "Hello World!";
  return 0;
}
```

- *cout << "Hello World!";* This line is a C++ statement.
  - *cout* is declared in the iostream standard file within the std namespace.
- *return 0;* causes the main function to finish.
  - interpreted as the program worked as expected without any errors during its execution.

Input and Output (cin and cout respectively)

- **Cout** is an object used for printing data to the screen.
  - The word cout followed by the insertion operator also called output redirection operator (<<) and the object to be printed on the screen.
  - *Syntax*: **Cout<<Object;**
  - The object at the right-hand side can be:
    - A literal string: "Hello World"
    - A variable: a place holder in memory
- **Cin** is an object used for taking input from the keyboard and store it in the memory.
  - The word cin followed by the input redirection operator (>>) and the object name to hold the input value.
  - *Syntax*: **Cin>>Object;**
- Both << and >> enabling multiple input or multiple output operations to be combined into one statement.

Keywords

- **Keywords/ reserved words** are standard identifiers and their functions is predefined by the compiler.
  - We cannot use keywords as variable names, class names, or method names, or as any other identifier.

  - ***Keywords***: asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while, and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq

Identifiers

- **Identifiers** is a name given to program elements such as variables, array, class and functions etc. It is a sequence of letters, digits, and underscores.
- Following rules must be kept in mind while naming an identifier.
  - The first character must be an alphabet (uppercase or lowercase) or can be an underscore.
  - An identifier can not start with a digit.
  - All succeeding characters must be alphabets or digits or underscore.
  - No special characters like !, @, #, $, % or punctuation symbols is allowed except the underscore"_".
  - No two successive underscores are allowed.
  - Keywords can not be used as identifiers.

Variables

- **Variable** is a named location in memory that is used to hold a value which may be modified by the program.
  - *Syntax:*

    datatype variable = initial;

  - *Example*:

```
int i; int m; int i, j, k;
char ch;
double current_balance;
int num_stud = 31; //variable initialization
```

…cont. Variables

- **Scope of variables** is the boundary or block in a program where a variable can be accessed.
  - **Global variables:** are variables that can be referred/accessed anywhere in the code, within any function, as long as it is declared first.
  - **Local Variables:** the scope of the local variable is limited to the code level or block within which they are declared.

```
#include<iostream.h>
int num1;
int add( int x, int y)
{
 int z;
 ….
}
void main()
{
 unsigned short age;
 float num2;
 cout<<"\n Enter your age:";
 …
}
```

Basic Data Types

- When you define a variable in C++, you must tell the compiler what kind of variable it is: an integer, a character, and so forth.
- This information tells the compiler how much room to set aside and what kind of value you want to store in your variable.
- Several data types are built into C++. The varieties of data types allow programmers to select the type appropriate to the needs of the applications being developed.
- Basic (fundamental) data types in C++ can be conveniently divided into numeric and character types

…cont. Basic Data Types

- **Numeric variables** can further be divided into integer variables and floating-point variables.
  - **Integer variables** will hold only integers.
  - **Floating number variables** can accommodate real numbers.

- Both the numeric data types offer modifiers that are used to vary the nature of the data to be stored.

- The **modifiers** used can be short, long, signed and unsigned.

…cont. Basic Data Types

- The data types used in C++ programs are described in following table

Type	Size	Value
unsigned short int	2 bytes	0 to 65,535
short int(signed short int)	2 bytes	-32,768 to 32,767
unsigned long int	4 bytes	0 to 4,294,967,295
long int(signed long int)	4 bytes	-2,147,483,648 to 2,147,483,647
int	2 bytes	-32,768 to 32,767
unsigned int	2 bytes	0 to 65,535
signed int	2 bytes	-32,768 to 32,767
Char	1 byte	256-character values
Float	4 bytes	3.4e-38 to 3.4e38
Double	8 bytes	1.7e-308 to 1.7e308
long double	10 bytes	1.2e-4932 to 1.2e4932
Bool	1 byte	True/false (top 7 bits are ignored)

…cont. Basic Data Types

- **Signed and Unsigned**
  - **Signed integers** are either negative or positive.
  - **Unsigned integers** are always positive.
  - *Example*: Lets us have only 4 bits to represent numbers.

Unsigned		Signed	
Binary	Decimal	Binary	Decimal
0 0 0 0	→0	0000	→0
0 0 0 1	→1	0001	→1
0 0 1 0	→2	0010	→2
0 0 1 1	→3	0011	→3
0 1 0 0	→4	0100	→4
0 1 0 1	→5	0101	→5
0 1 1 0	→6	0110	→6
0 1 1 1	→7	0111	→7
1 0 0 0	→8	1000	→0
1 0 0 1	→9	1001	→ -1
1 0 1 0	→10	1010	→ -2
1 0 1 1	→11	1011	→ -3
1 1 0 0	→12	1100	→ -4
1 1 0 1	→13	1101	→ -5
1 1 1 0	→14	1110	→ -6
1 1 1 1	→15	1111	→ -7

Characters

- **Characters variables** (type char) are typically one byte in size, enough to hold 256 different values.
- Char in C++ are represented as any value inside a single quote.
  - *Example*: ‘x’, ‘A’, ‘5’, ‘a’, etc.
- When the compiler finds such values (characters), it translates back the value to the ASCII values.
  - *Example*: ‘a’ has a value 97 in ASCII.
- Some special characters used for formatting.

\n	new line
\t	tab
\b	backspace
\”	double quote
\’	single quote
\?	question mark
****	backslash

Constants

- **Constant** is any expression that has a fixed value whose values cannot be modified or changed during the execution of program. Also called as ***Literals***.
  - Constant must be initialized when declared as values cannot be assigned it to later.
  - **Literal constant:** is a value typed directly into the program wherever it is needed.
    - *Example*: **int num = 43;**
    - 43 is a literal constant in this statement:
  - **Symbolic constant:** is a constant that is represented by a name, similar to that of a variable. But unlike a variable, its value can't be changed after initialization.
    - *Example*: **int availableStudentInClass =25;**

…cont. Constants

- **Enumerated Constants** is used to declare multiple integer constants using a single line with different features.
- The enum (short form of enumerated) type cannot take any other data type than integer.
- The first constant will have the value 0, and the values for the remaining constants will count up from the initial value by 1.
- Every enumerated constant has an integer value.
  - *Example*: **enum** SEASON**{** SPRING, SUMMER, FALL, WINTER **}**;

- But one can also assign different numbers for each.
  - *Example*: **enum** COLOR**{** RED=100,BLUE,GREEN=500,WHITE,BLACK **}**;

Comments

- **Comments** are for the reader, not the compiler
- Two types:
  - **Single line**

```
// This is a C++ program. It prints the sentence:
// Welcome to C++ Programming.
```

  - **Multiple line**

```
/*
You can include comments that can
 occupy several lines.
*/

/* Welcome to C++ Programming. */
```

Operators

- **Operator** is a special symbol that is used to carry out some specific operation on its operand.
  - Assignment operator
  - Arithmetic operator
  - Relational operator
  - Logical operator
  - Bitwise operator
  - Increment/decrement operator
  - Conditional operator
  - Comma operator
  - The size of operator
  - Explicit type casting operators, etc

...cont. Operators

- **Assignment operator (=):** is used to assign value to a variable, you can assign a variable value or the result of an arithmetical expression.

- *Syntax*: **Operand1 = Operand2;**

  - **Operand1** is always a variable
  - **Operand2** can be one or combination of:

    - **Literal constant (***example*: **x=12;)**
    - **Variable (***example*: **x=y;)**
    - **Expression (***example*: **x=y+2;)**

...cont. Operators

- **Assignment operators**

Operator	Description	Expression
=	Assignment Operator	a=b
+=	add and assign	a+=b is equivalent to a=a + b
-=	subtract and assign	a-=b is equivalent to a=a - b
*=	multiply and assign	a*=b is equivalent to a=a * b
/=	divide and assign	a/=b is equivalent to a=a/b
%=	mod and assign	a%=b is equivalent to a=a % b
<<=	Left shift AND assign	a<<=5 is equivalent to a=a<<5
>>=	Right shift AND assign	a>>=5 is equivalent to a=a>>5
&=	Bitwise AND assign	a&=5 is equivalent to a=a&5
^=	Bitwise exclusive OR and assign	a^=5 is equivalent to a=a^5
|=	Bitwise inclusive OR and assign	a|=5 is equivalent to a=a|5

...cont. Operators

- **Arithmetic operator:** is used to perform arithmetic operations like addition, subtraction, multiplication, division, %modulus, exponent, etc.
- Let variable a holds 20 and variable b holds 10, then

Operator	Name	Description	Example
+	Addition	Addition of given operands	**a + b** //returns 30
-	Subtraction	Subtraction of second operand from first	**a - b** //returns 10
*	Multiplication	Multiplication of given operands	**a * b** //returns 200
/	Division	Returns Quotient after division	**a / b** //returns 2
%	Modulus	Returns Remainder after division	**a % b** //returns 0

...cont. Operators

- **Increment/decrement operator:** In C++, ++ and — are know as increment and decrement operators respectively.
  - ++ adds 1 to operand and — subtracts 1 to operand respectively.
  - *Example*: int k = 5;

Operator	Example	Description	Example
++ [prefix]	++a	The value of a after increment	**++k + 10** // gives 16
++ [postfix]	a++	The value of a before increment	**k++ + 10** // gives 15
— [prefix]	–a	The value of a after decrement	**--k + 10** // gives 14
— [postfix]	a–	The value of a before decrement	**--k + 1** // gives 14

Operators

- **Comparison Operators:** are used evaluate a comparison between two operands.
  - The result is a Boolean value that can only be true or false.
  - Also referred as relational operators.
  - Let variable **a** holds **20** and variable **b** holds **10**, then

Operator	Description	Example
>	greater than	a>b returns TRUE
<	Less than	a<b returns FALSE
>=	greater than or equal to	a>=b returns TRUE
<=	less than or equal to	a<=b returns FALSE
==	is equal to	a==b returns FALSE
!=	not equal to	a!=b returns TRUE

...cont. Operators

- **Logical operators** are used to combine expressions with conditional statements using logical (AND, OR, NOT) operators, which results in true or false.
  - Let variable **a** holds **true** or **1** and variable **b** holds **false** or **0**, then

Operator	Name	Description	Example
&&	Logical AND	Return true if all expression are true	(a && b) //returns false
||	Logical OR	Return true if any expression is true	(a || b) //returns true
!	Logical NOT	Return complement of expression	!a //returns false

Operators

- **Conditional / Ternary operator (? :)** is considered as short hand for if-else statement.
  - *Syntax*: **condition ? result1 : result2**;
  - If condition is true the expression will return result1, if it is not it will return result2.
  - *Example*: result = (**10 > 15**) **?** "**Greater**" **:** "**Smaller**";
    - *Output*: **Smaller**

...cont. Operators

- **Bitwise operator** is for manipulating the individual bits in an integer quantity.

- Bitwise operators types are:
  - **Bitwise negation** is a unary operator which reverses the bits in its operands.
  - **Bitwise and** compares the corresponding bits of its operands and produces a 1 when both bits are 1, and 0 otherwise.
  - **Bitwise or** compares the corresponding bits of its operands and produces a 0 when both bits are 0, and 1 otherwise.
  - **Bitwise exclusive or** compares the corresponding bits of its operands and produces a 0 when both bits are 1 or both bits are 0, and 1 otherwise.

Operators

- **Left shifting** of an integer "**x**" with an integer "**y**" denoted as '**(x<<y)'** is equivalent to multiplying **x** with **2^y** (2 raised to power y).
  - *Example*: lets take **N=22**; which is **00010110** in Binary Form.
  - Now, if "*N is left-shifted by 2*" i.e **N=N<<2** then **N** will become **N=N*(2^2).**
  - Thus, **N=22*(2^2)=88** which can be written as **01011000.**
- **Right shifting** an integer "**x**" with an integer "**y**" denoted as '**(x>>y)**' is equivalent to dividing x with 2^y.
  - *Example***:** lets take **N=32**; which is **100000** in Binary Form.
  - Now, if "*N is right-shifted by 2*" **i.e N=N>>2** then **N** will become **N=N/(2^2).**
  - Thus, **N=32/(2^2)=8** which can be written as **1000**.

Operators

- Binary equivalent of given numbers:
  - **10 = 00 1010**
  - **11 = 00 1011**
  - **12 = 00 1100**

Operator	Name	Example	
~	Bitwise Negation	~11	// gives -12
&	Bitwise And	10 & 12	// gives 8
|	Bitwise Or	10 | 12	// gives 14
^	Bitwise Exclusive Or	10 ^ 12	// gives 6
<<	Bitwise Left Shift	10 << 2	// gives 40
>>	Bitwise Right Shift	10 >> 2	// gives 2

...cont. Operators

- **sizeof() operator** returns the size of its operand's data type in bytes.
  - *Example*: a = **sizeof** (**char**);
    - ***Output***: a = 1 because char is a one-byte long data type.
- **Comma operator** (,) allows us to separate two or more expressions that are included where only one expression is expected.
  - *Example*: **i = (j=3, j+2); int i, j, k;**
    - First assign the value 3 to j then assign j+2 to variable i.
    - ***Output***: i=5
- **Type casting operators** allows you to convert a datum of a given type to another data type.
  - *Example*: int i; float f = 3.14; i = (int)f; equivalent to i = int(f);
    - value variable i is 3 - ignoring the decimal point.

Operator Precedence

- **Operator precedence** defines the order in which given mathematical expression is evaluated.
  - **Left-associative**- are evaluated from the left to right,
  - **Right-associative**- are evaluated from right to the left.
  - **No associativity**- does not follow any predefined order.

Level	Operator	Order
Highest	**sizeof() ++ -- (pre fix)**	Right to left
	*** / %**	Left to right
	+ -	Left to right
	< <= > >=	Left to right
	== !=	Left to right
	&&	Left to right
	||	Left to right
	? :	Left to right
	= ,+=, -=, *=, /=,^= ,%=, &= ,|= ,<<= ,>>=	Right to left
	++ -- (postfix)	Left to right
	,	Left to right

Chapter Three

Control Statements

Outlines

- Introduction
- Selection Statements
  - If
  - If...else
  - Nested if.... else
  - Switch
- Jump statements
  - Break
  - Continue
  - Go to
- Repetition statements (loops)
  - For loop
  - While loop
  - Do... while loop
  - Nested loops
  - Infinite loop

Introduction

- Control flow or flow of control is the order in which instructions, statements and function calls being executed or evaluated when a program is running.
- The control flow statements are also called as Flow Control Statements.
- In C++, control flow statements are used to alter, redirect, or to control the flow of program execution based on the application logic.
- In C++, Control flow statements are mainly categorized in following types:
  - Selection statements
  - Iteration statements
  - Jump statements

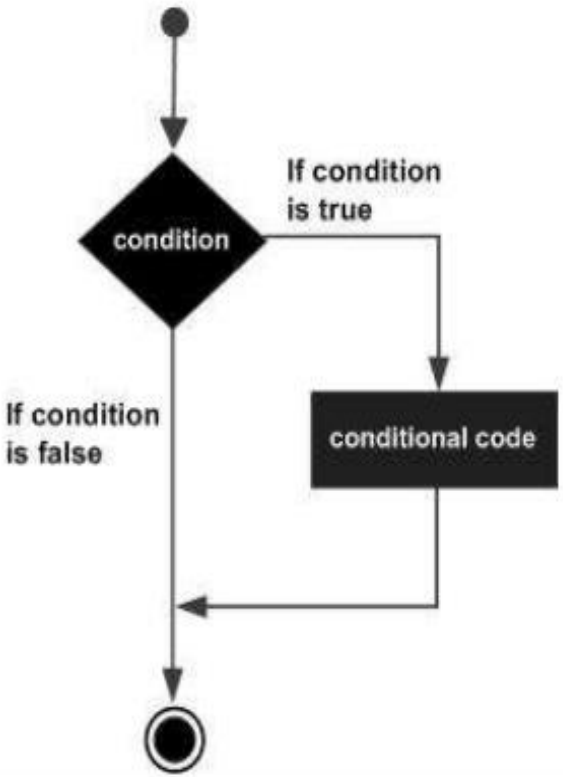
C++ Selection Statements
if statement

- If statement consists of a boolean expression followed by one or more statements.
- **Syntax**:

```
if(boolean_expression) {
    // statement(s) will execute if the boolean expression is true
}
```

- If the boolean expression evaluates to true, then the block of code inside the if statement will be executed.
- If boolean expression evaluates to false, then the first set of code after the end of the if statement (after the closing curly brace) will be executed.

C++ Selection Statements
if statement




```
#include <iostream>
using namespace std;
int main(){
    int a,b;
    cout<<"Enter two numbers to compare?"<<endl;
    cin>>a>>b;
    if(a>b){
        cout<< a<<" is greater than "<<b<<".";
    }
return 0;
}
```

- If you enter 55 for a and 10 for b, the **output** is:

55 is greater than 10.

- If you enter 5 for a and 10 for b, **no output**

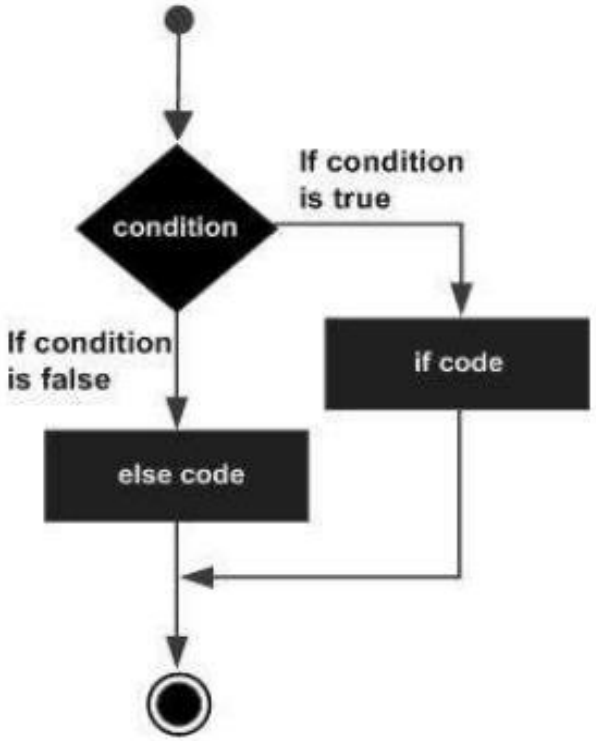
C++ Selection Statements
if…else statement

- In C++, when we want to execute a block of code when if condition is true and another block of code when if condition is false.
- **Syntax**:

```
if (condition) {
        // statement(s) will execute if the condition is true
}
else {
        // statement(s) will execute if the condition is false
}
```

- If the condition evaluates to **true**, then the **if block** of code will be executed, otherwise **else block** of code will be executed.

C++ Selection Statements
if…else statement




```
#include <iostream>
using namespace std;
int main(){
    int a,b;
    cout<<"Enter two numbers to compare?"<<endl;
    cin>>a>>b;
    if(a>b){
        cout<< a<<" is greater than "<<b<<".";
    }else{
        cout<<a<<" is less than "<<b<<". ";
    }
return 0;
}
```

- If you enter 55 for a and 10 for b, the **output** is:

55 is greater than 10.

- If you enter 5 for a and 10 for b, the **output** is:

5 is less than 10.

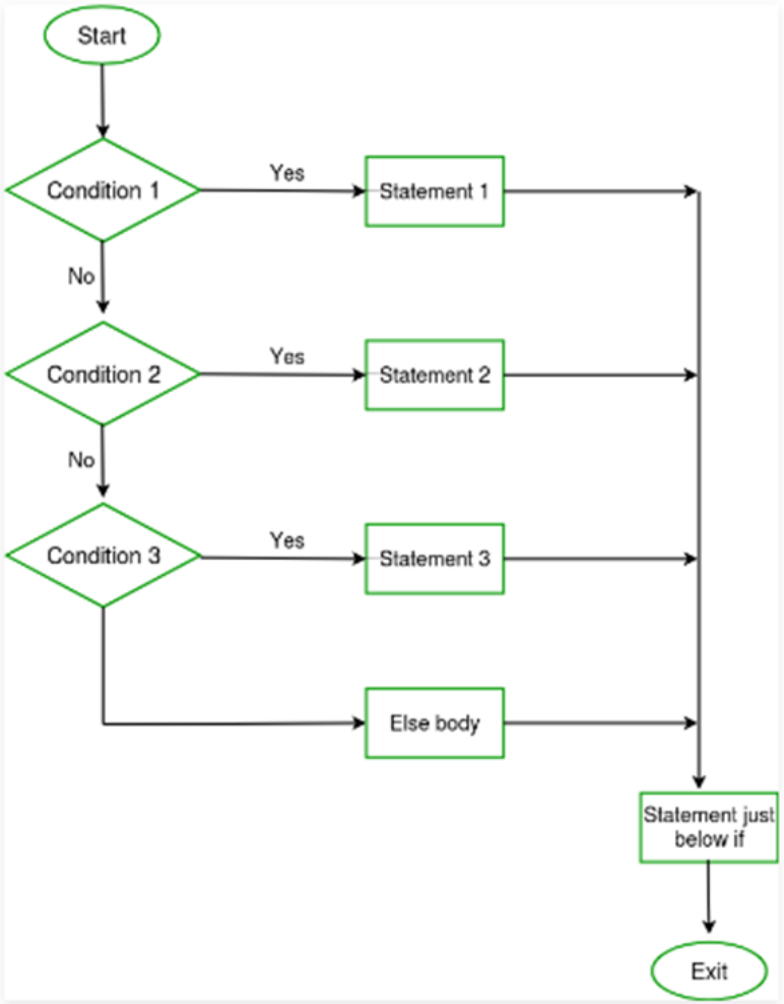
C++ Selection Statements
if...else if...else statement

- An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions.
- **Syntax**:

```
if (condition_1) {
        // statement(s) will execute if the condition_1 is true
}
else if (condition_2) {
        // statement(s) will execute if the condition_2 is true
}
else {
        // statement(s) will execute if the condition_3 is false
}
```

C++ Selection Statements

if...else if...else statement




```
// C++ program to illustrate if-else-if
#include<iostream>
using namespace std;

int main()
{
    int i = 20;

    if (i == 10)
        cout<<"i is 10";
    else if (i == 15)
        cout<<"i is 15";
    else if (i == 20)
        cout<<"i is 20";
    else
        cout<<"i is not present";
}
```

Output:

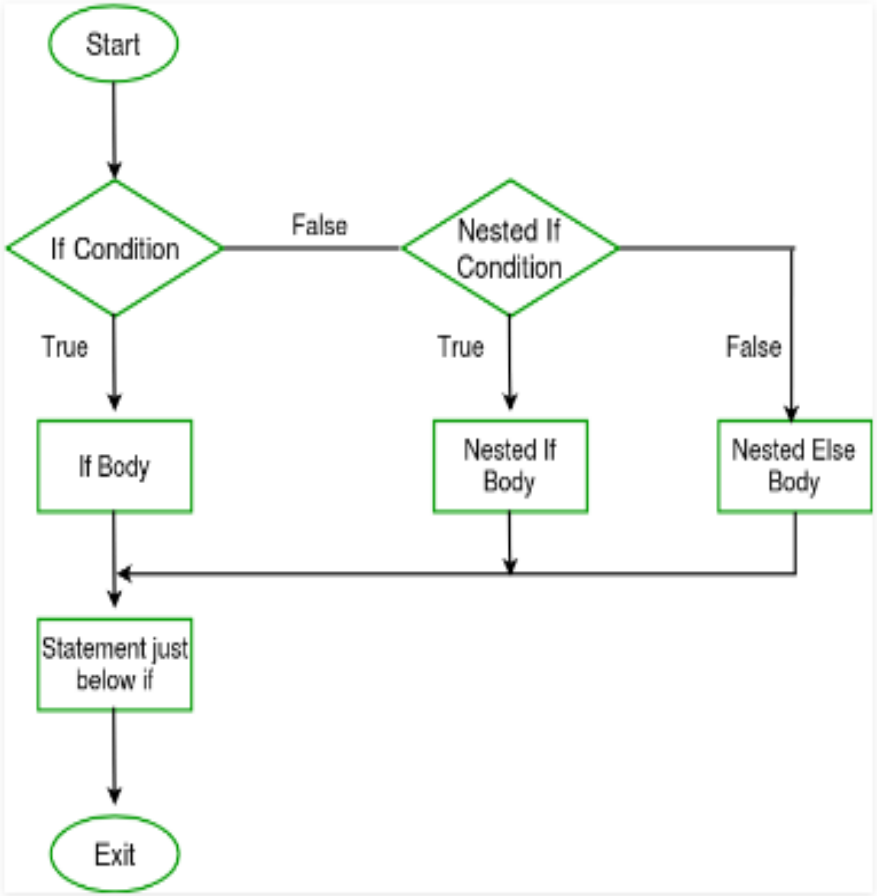
i is 20

C++ Selection Statements
nested if statement

- Nested if statements means an if statement inside another if statement.
- **Syntax**:

```
if (condition_1){
    // Executes when condition_1 is true
    if (condition_2){
        // Executes when condition_1 and then condition_2 is true
    }
}
```

C++ Selection Statements
nested if statement




```cpp
// C++ program to illustrate nested-if statement
#include <iostream>
using namespace std;

int main()
{
    int i = 10;

    if (i == 10)
    {
        // First if statement
        if (i < 15)
           cout<<"i is smaller than 15\n";

        // Nested - if statement
        // Will only be executed if statement above
        // is true
        if (i < 12)
            cout<<"i is smaller than 12 too\n";
        else
            cout<<"i is greater than 15";
    }

    return 0;
}
```

Output:

i is smaller than 15

i is smaller than 12 too

C++ Switch Case Statement

- In C++, switch case statement is simplified form of the C++ Nested if else statement , it helps to avoid long chain of if...else if...else statements.
- A switch case statement evaluates an expression against multiple cases in order to identify the block of code to be executed.

- Syntax:

```
switch (expression){
case value1:
    // statements
    break;
case value2:
    // statements
    break;
default:
    // statements
    break;
}
```

C++ Switch Case Statement

```
#include <iostream>
using namespace std;
int main(){
    const int dayOfWeek = 5;
    cout<<"C++ Switch Case statement.";
    switch(dayOfWeek){
      case 1:
        cout<<"\nToday is Monday.";
        break;
      case 2:
        cout<<"\nToday is Tuesday.";
        break;
      case 3:
        cout<<"\nToday is Wednesday.";
        break;
      case 4:
        cout<<"\nToday is Thursday.";
        break;
      case 5:
        cout<<"\nToday is Friday.";
        break;
      case 6:
        cout<<"\nToday is Saturday.";
        break;
      case 7:
        cout<<"\nToday is Sunday.";
        break;
      default:
        cout<<"\nInvalid Weekday.";
        break;
    }
    return ;
}
```

Output:

C++ Switch Case statement.
Today is Friday.

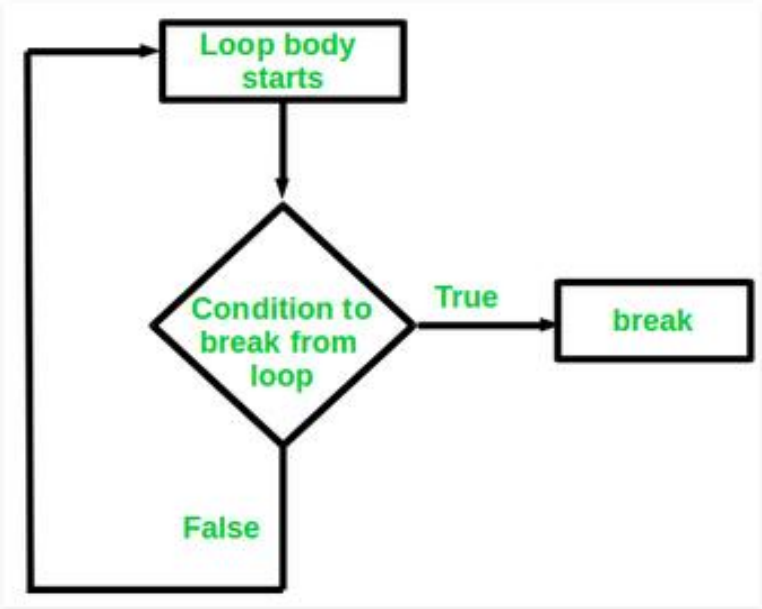
Jump Statements
Introduction

- **Loop control statements change execution from its normal sequence.**
- **Break statement**
  - Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.
- **Continue statement**
  - Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
- **Go to statement**
  - Transfers control to the labelled statement. Though it is not advised to use go to statement in your program.

Jump Statements
break

- **Break** is a type of loop control statement which is used to terminate the loop.
- **Syntax:**

  break;




```
// C++ program to illustrate
#include <iostream>
using namespace std;
int main(){
    // the sum of 1-5
    int sum=0;
    for (int i=1;i<15;i++){
        if i<=5{
          sum += i;
        }
        else{
          break;
        }
    }
    cout<<"The sum of 1 upto 5 is "<<sum;
    return 0;
}
```

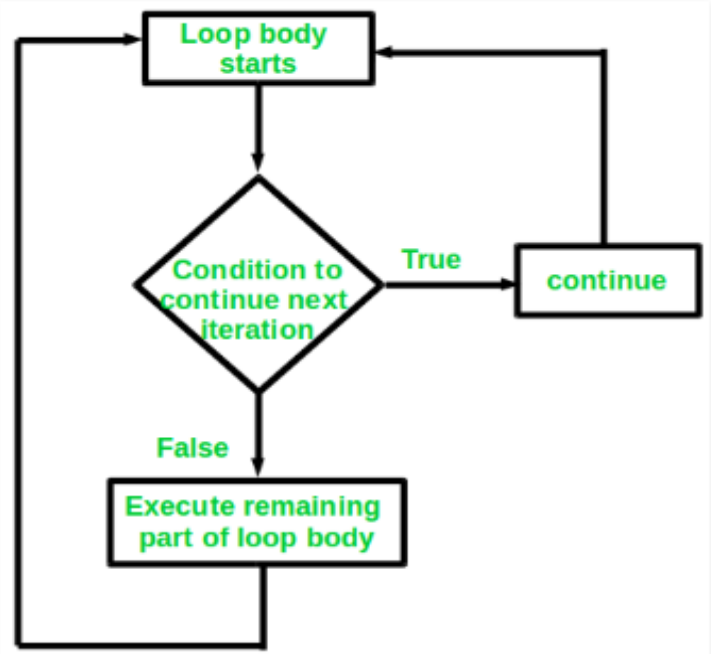
Output:

The sum of 1 upto 5 is 15

Jump Statements
continue

- The continue statement is opposite to that of break statement, instead of terminating the loop, it forces to continue or execute the next iteration of the loop.
- When the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped and next iteration of the loop will begin.
- **Syntax:**

```
continue;
```



Jump Statements

continue

```
/* C++ program to explain the use
of continue statement */
#include <iostream>
using namespace std;
int main(){
    // loop from 1 to 10
    for (int i = 1; i <= 10; i++) {
        // If i is equals to 6,
        // continue to next iteration
        // without printing
        if (i == 6)
            continue;
        else
            // otherwise print the value of i
            cout << i << " ";
    }
    return 0;
}
```

Output:

1 2 3 4 5 7 8 9 10

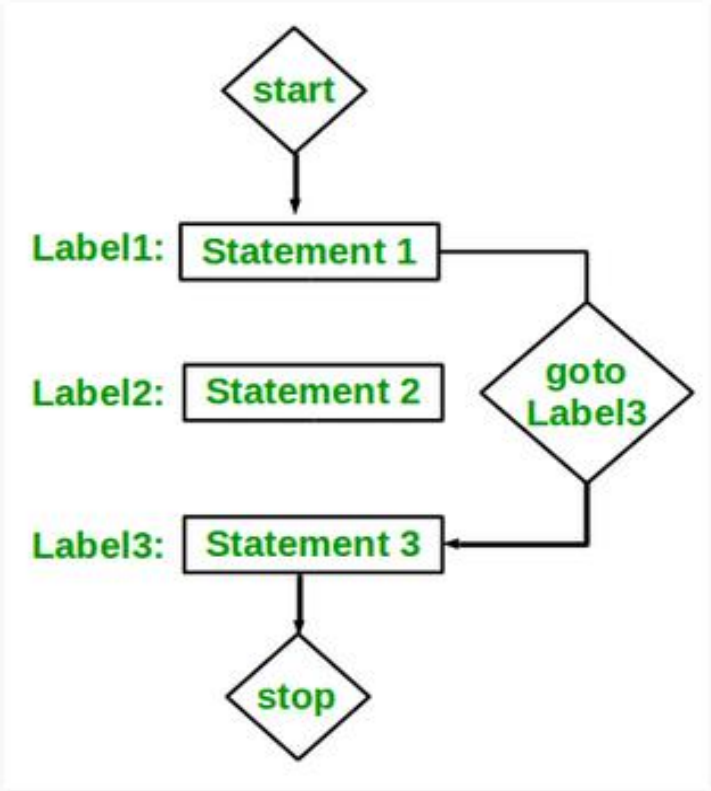
Jump Statements

goto

- The goto statement in C/C++ also referred to as unconditional jump statement can be used to jump from one point to another within a function.

- **Syntax:**

Syntax1	Syntax2
goto label;	label:
.	.
.	.
.	.
label:	goto label;



Jump Statements

goto

```
#include <iostream>
using namespace std;
// function to print numbers from 1 to 10
void printNumbers(){
    int n = 1;
    label:
      cout << n << " ";
      n++;
    if (n <= 10){
        goto label;
    }
}
// Driver program to test above function
int main(){
    printNumbers();
    return 0;
}
```

Output:

1 2 3 4 5 6 7 8 9 10

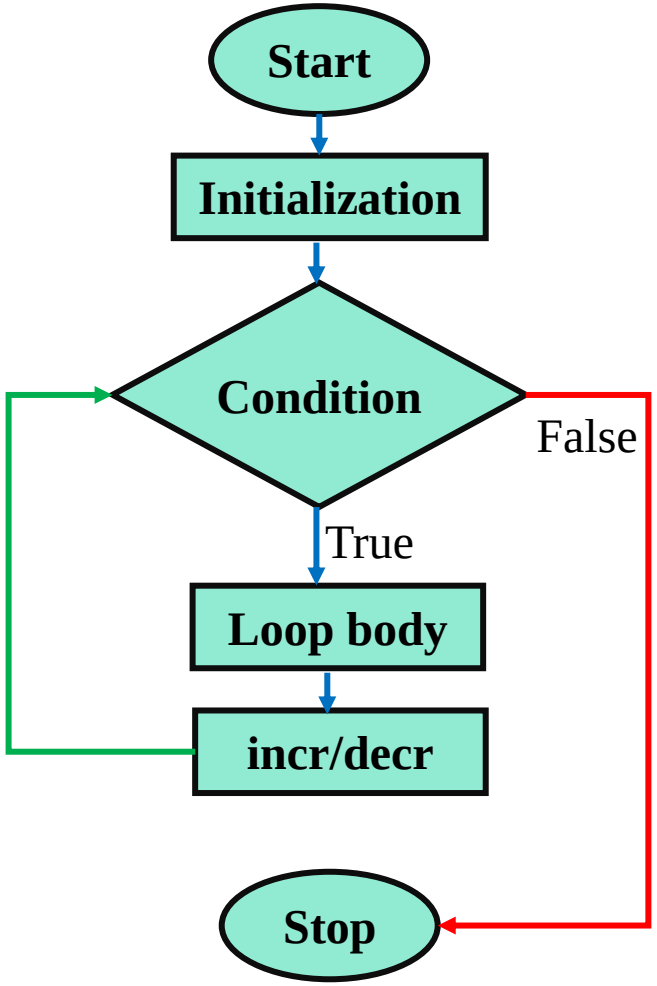
Looping Statements

- Loop statements are used to execute the block of code repeatedly for a specified number of times or until it meets a specified condition.
- Loop statement are very useful to iterate over collection/list of items or to perform a task for multiple times.
- In C++, we have following loop statements available:
  - For loop
  - While loop
  - Do...while loop

Looping Statements
for loop

- The for loop is used when we want to execute block of code known times. The for loop takes a variable as iterator and assign it with an initial value, and iterate through the loop body as long as the test condition is true.
- Syntax:

```
for(initialization; condition; incr/decr){
        // loop body
        // Or statement(s)
}
```



Looping Statements
for loop

- **initialization**: this section is used to declare and initialize the loop iterator.
- **condition**: if the condition returns a true value then the loop body is executed for current iteration. If it returns a false value then loop is terminated and control jumps to the next statement just after the for loop.
- **incr/decr**: Once, the loop body is executed for current iteration then the incr/decr statement is executed to update the iterator value and the test condition is evaluated again.

```
#include <iostream>
using namespace std;
int main ()
{
 // Local variable declaration:
 int a = 15;
 // for loop execution
     for(a; a < 20; a++)
     {
         cout << "value of a: " << a << endl;
     }
 return 0;
}
```

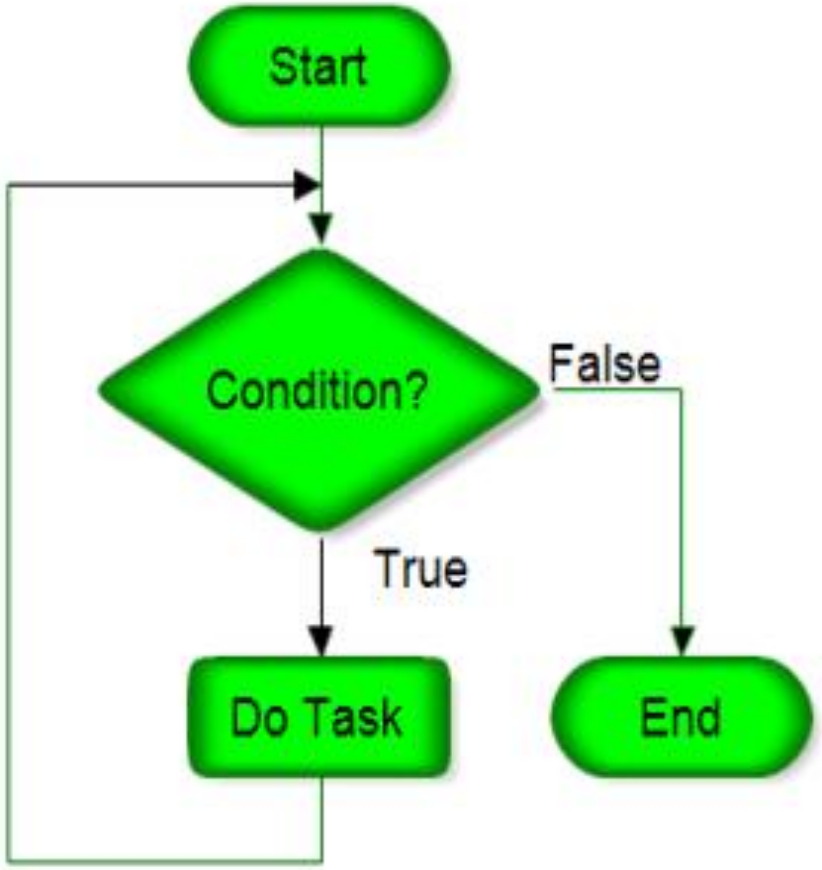
```
The output produces the following result:
    value of a: 15
    value of a: 16
    value of a: 17
    value of a: 18
    value of a: 19
```

Looping Statements
while loop

- The while loop will execute a block of statement as long as a test expression is true.

- **Syntax:**

```
while(condition){
    // loop body
}
```



Looping Statements

while loop

```
#include <iostream>
using namespace std;
int main ()
{
 // Local variable declaration:
 int a = 15;
 // while loop execution
      while( a < 20 )
      {
          cout << "value of a: " << a << endl;
          a++;
      }
 return 0;
}
```

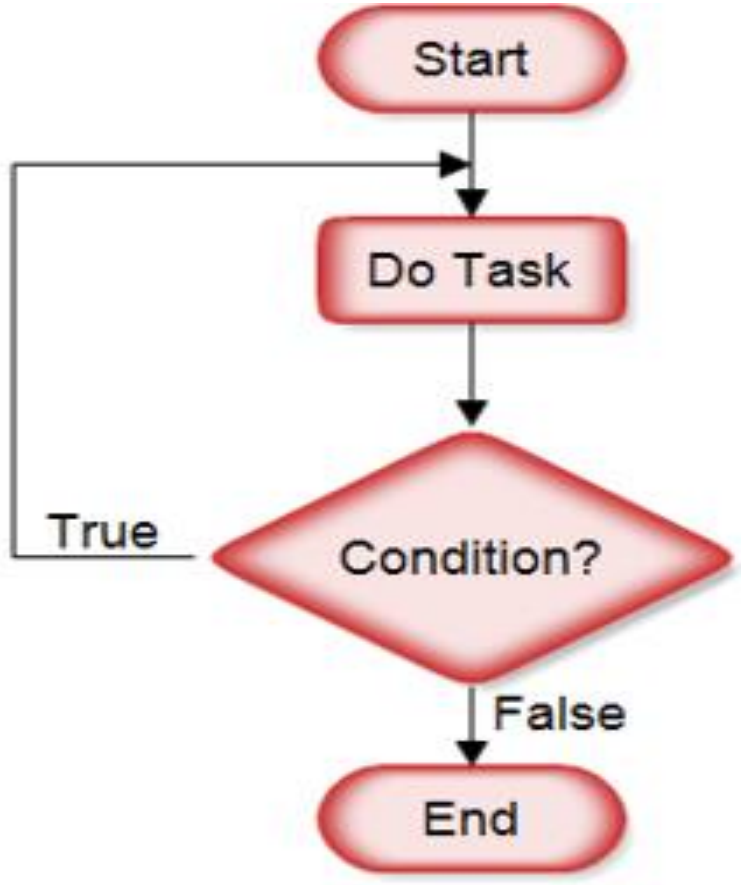
The output produces the following result:

```
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Looping Statements

do…while loop

- The do...while statement executes loop statements and then test the condition for next iteration and executes next only if condition is true.



- Syntax:

```
do{
   // loop body
} while(condition);
```

Looping Statements

do…while loop

```
#include <iostream>
using namespace std;
int main ()
{
 // Local variable declaration:
 int a = 15;
 // do...while loop execution
      do
      {
          cout << "value of a: " << a << endl;
          a++;
      }while(a<20);
 return 0;
}
```

The output produces the following result:

```
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Looping Statements
infinite loop

- A loop becomes infinite loop if a condition never becomes false. The for loop is traditionally used for this purpose. Since none of the three expressions that form the 'for' loop are required, you can make an endless loop by leaving the conditional expression empty.
- When the conditional expression is absent, it is assumed to be true.
- Terminate an infinite loop by pressing Ctrl + C keys.

- Example:

```
#include <iostream>
using namespace std;
int main () {
    for( ; ; ) {
        cout<<"This loop will run forever. \n";
    }
return 0;
}
```

Chapter Four

C++ Functions

Outlines

- Introduction
- User Defined Functions
- Function Prototype and Return Statement
- Integration of a Function
- Functions with Empty Parameter List
- Function Overloading
- The inline Functions
- C++ Standard Library Functions
- Passing Arguments by Value, by Reference, and by Pointer
- Recursive Functions

Introduction

- A **function** in C++ is a group of statements that together perform a specific task.
- Every C/C++ program has at least one function that the name is **main**.
- The main function is called by the **operating system** by which our code is executed.
- We can make n number of function in a single program but we can make only one main function in a single program.
- Every program has only one main function.

- There are two types of functions:
  - **Standard Library Functions:** Predefined in C++
  - **User-defined Function:** Created by users

User-defined Function

- C++ User-defined Function. C++ allows the programmer to define their own function.

- A user-defined function groups code to perform a specific task and that group of code is given a name (identifier).

- When the function is invoked from any part of the program, it all executes the codes defined in the body of the function.

- Syntax:

```
returnType functionName (parameter1, parameter2,...) {
        // function body
}
```

User-defined Function

- **Return Type** – A function may return a value. The **return type** is the data type of the value the function returns.
  - Some functions perform the desired operations without returning a value.
    - In this case, the return type is the keyword **void**.
- **Function Name** – This is the actual name of the function. The function name and the parameter list together constitute the **function signature**.
- **Parameters** – A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as **actual parameter or argument**.
  - The **parameter list** refers to the type, order, and number of the parameters of a function.
  - Parameters are **optional**; that is, a function may contain no parameters.
- **Function Body** – The function body contains a collection of statements that define what the function does.

User-defined Function: *Function Declaration*

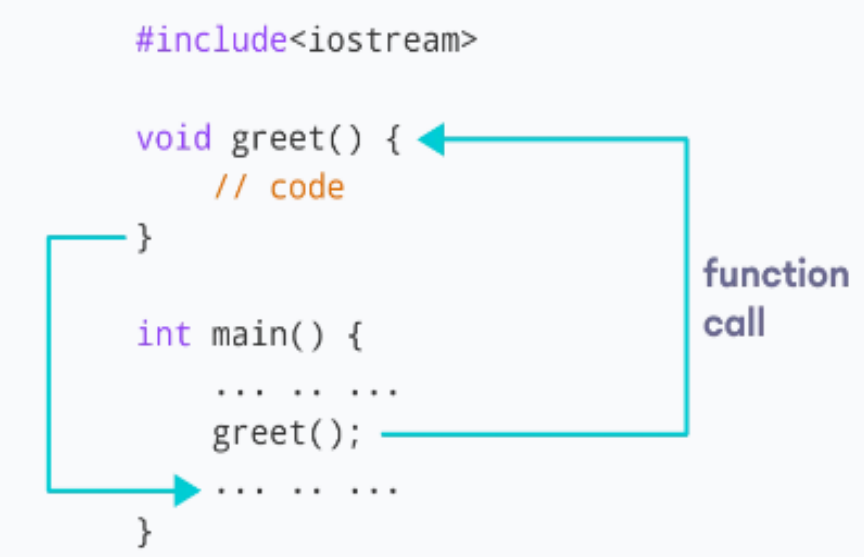
- A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.
- **Syntax**:

  returnType functionName (parameter list) ;
- **Example**: int max (int num1, int num2) ;
- Parameter names are not important in function declaration only their type is required.

  int max (int, int) ;
- Function declaration is required when you define a function in one source file and you call that function in another file.
  - In such case, you should declare the function at the top of the file calling the function.

User-defined Function: *Function Calling*

- To use a defined function, we have to call or invoke that function.
- When a program calls a function, program control is transferred to the called function.
- A called function performs defined task and when it's return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.



The 4 ways of calling a function

1. FunctionName();
2. FuctionName(arguments);
3. VarName = FuctionName();
4. VarName = FuctionName(args);

User-defined Function: *Function Calling*

1. Function not take any arguments and not return any value

```
void FunctionName (void){
   // Function Body
}
```

2. Function takes an argument and not return any value

```
void FunctionName (args){
   // Function Body
}
```

3. Function is take nothing and return something type

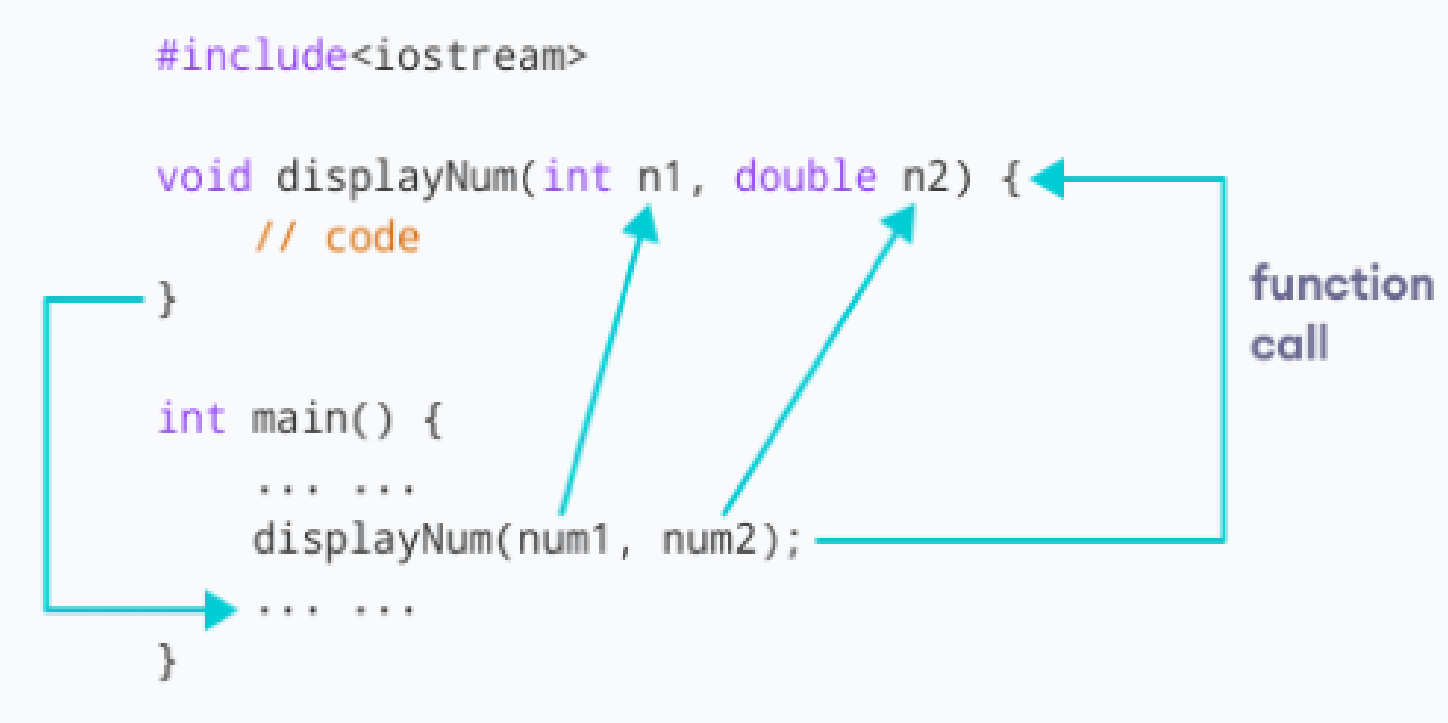
```
Ret_Type FunctionName (void){
   // Function Body
}
```

4. Function is take something and return something type

```
Ret_Type FunctionName (args){
   // Function Body
}
```

User-defined Function: *Function Calling*

- **Function Parameters/Arguments**
  - The type of the arguments passed while calling the function must match with the corresponding parameters defined in the function declaration.



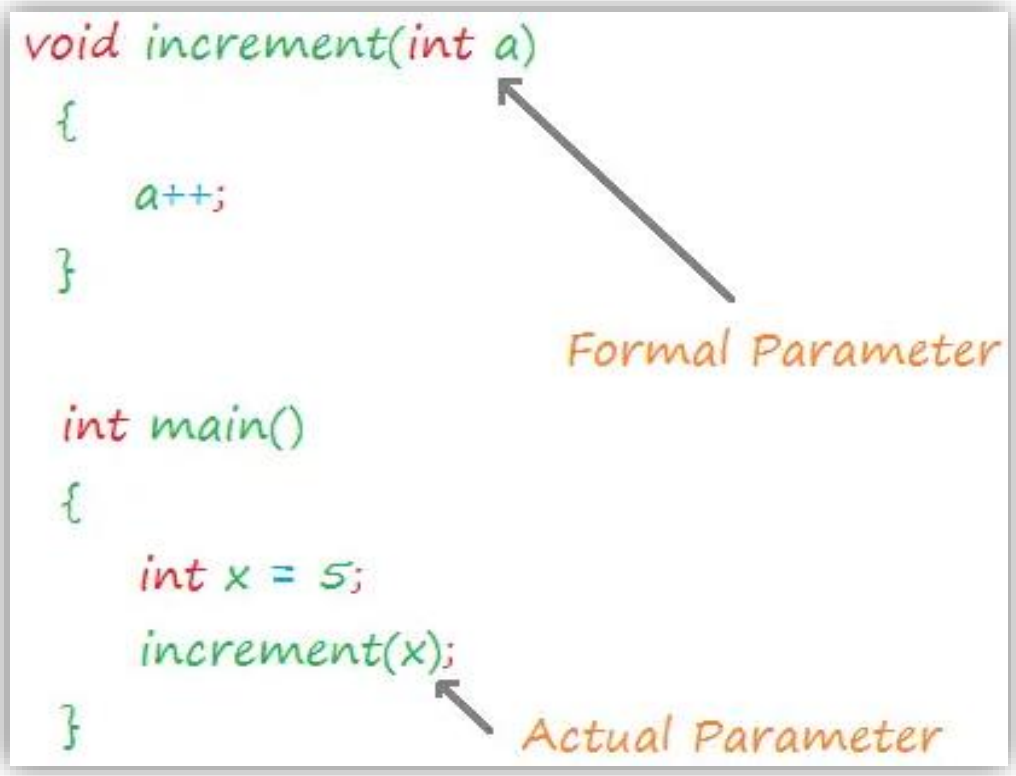
User-defined Function: *Function Calling*

- **Function Parameters/Arguments:** the three ways that arguments can be passed to a function.

No	Call Type	Description
1	Call by Value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	Call by Pointer	This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
3	Call by Reference	This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

User-defined Function: *Function Calling*

- **Actual Parameters** are the parameters that appear in the function call statement.
- **Formal Parameters** are the parameters that appear in the declaration of the function which has been called.



User-defined Function: *Function Calling*

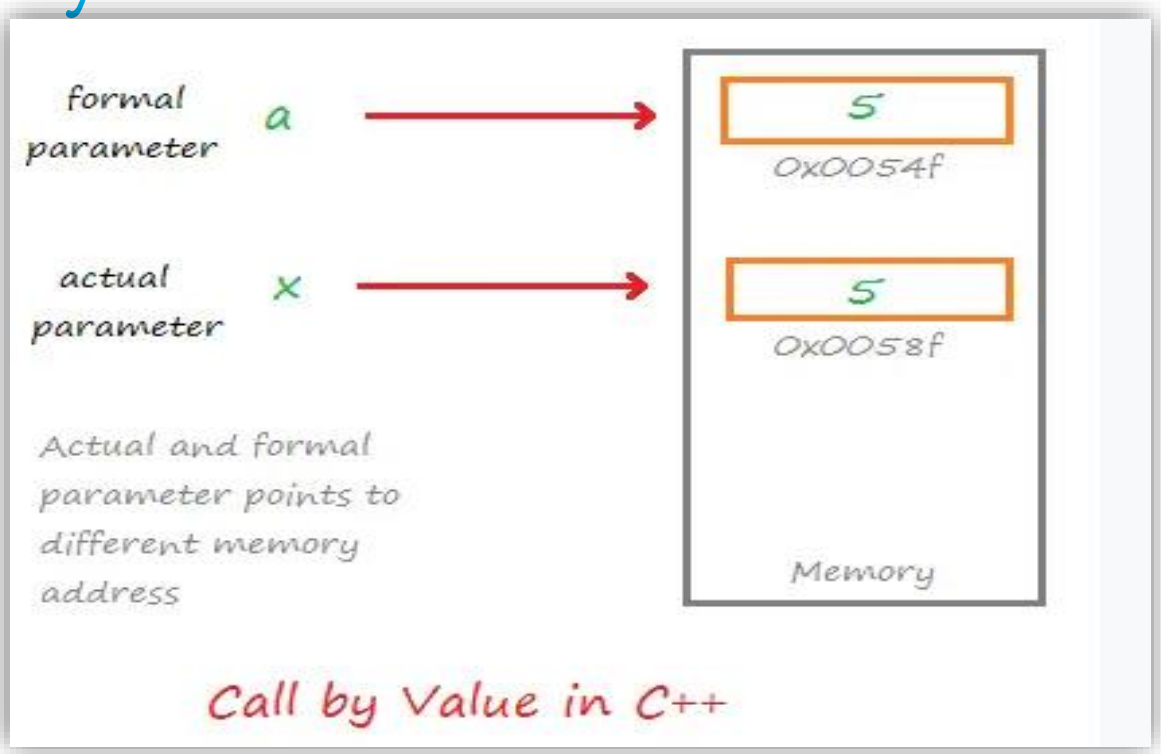
- **Call by Value:** When a function is called in the call by value, the **value of the actual parameters is copied into formal parameters**.

  - Both the actual and formal parameters **have their own copies of values**, therefore any change in one of the types of parameters will **not be reflected** by the other.

  - This is because both actual and formal parameters **point to different locations in memory** (i.e. they both have **different memory addresses**).

User-defined Function: *Function Calling*

- **Call by Value:**



- It is useful when we do not want the values of the actual parameters to be changed by the function that has been invoked.

User-defined Function: *Function Calling*

- *Call by Value:* Example-

```
#include <iostream>
using namespace std;

//Value of x gets copied into a
void increment(int a){
    a++;
    cout << "Value in Function increment: "<< a <<endl;
}

int main()
{
    int x = 5;
    increment(x);
    cout << "Value in Function main: "<< x <<endl;
    return 0;
}
```

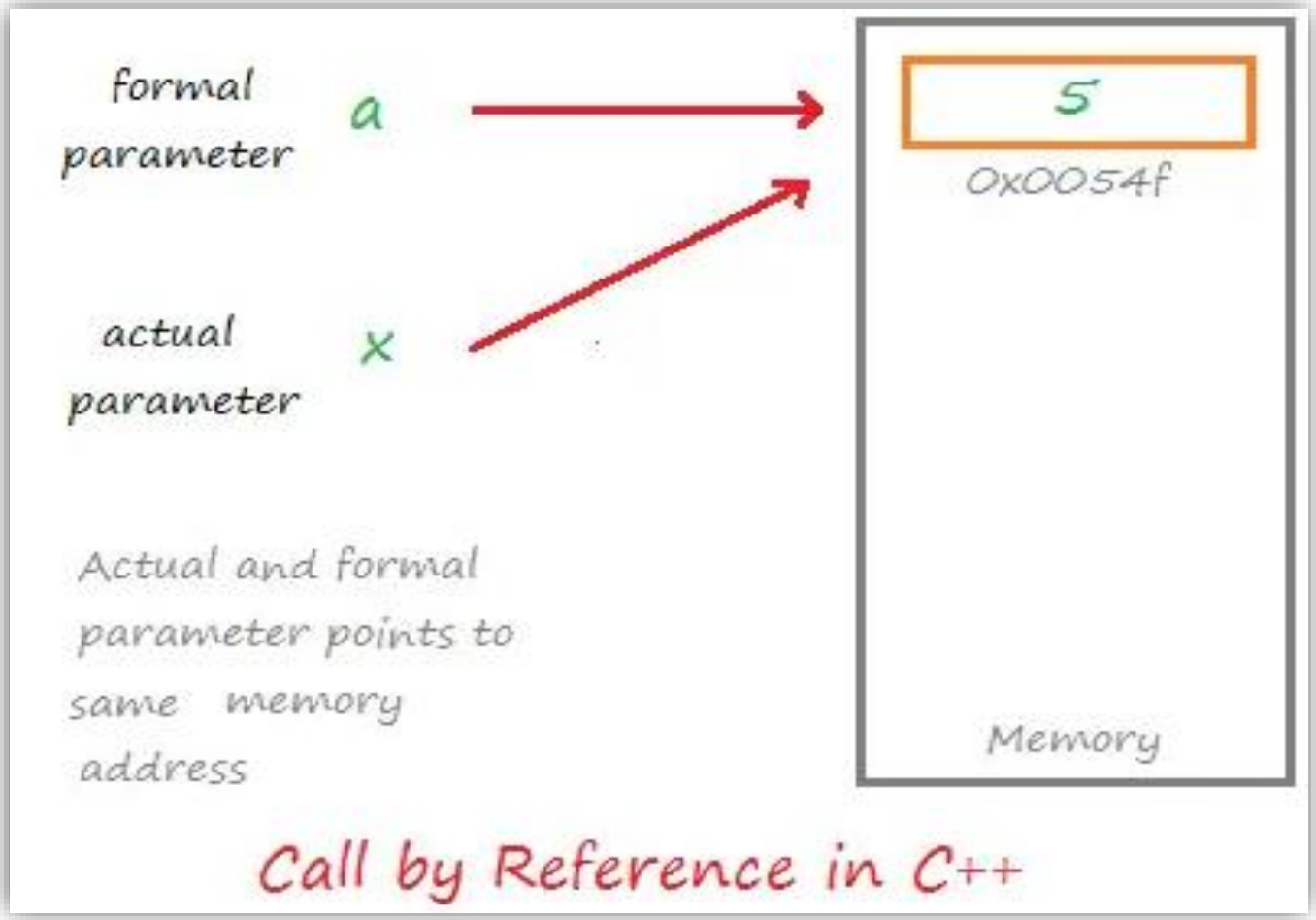
```
C:\Users\pc\Documents\Proje
Value in Function increment: 6
Value in Function main: 5

Process returned 0   execution time : 0.140 s
Press any key to continue.
```

User-defined Function: *Function Calling*

- **Call by Reference:** In the call by reference, **both formal and actual parameters share the same value.**
  - Both the actual and formal parameter points to the **same address in the memory**.
  - That means any change on one type of parameter will also **be reflected by other**.
  - Calls by reference are preferred in cases where we do not want to make copies of objects or variables, but rather we want all operations to be performed on the same copy.

User-defined Function: *Function Calling*

- Call by Reference:



User-defined Function: *Function Calling*

- Call by Reference: **Example-**
  - **Note:** for creating reference, the '**&**' operator is used in preceding of variable name.
  - **Note:** the output in this case. The value of 'a' is increased to 6, the value of 'x' in the main also changes to 6.
  - This proves that **changes made to formal parameters are also reflected by the actual parameters** as they share the same memory address space.

User-defined Function: *Function Calling*

- Call by Reference: Example-

```
#include <iostream>
using namespace std;

//Value of x is shared with a
void increment(int &a){
    a++;
    cout << "Value in Function increment: "<< a <<endl;
}

int main()
{
    int x = 5;
    increment(x);
    cout << "Value in Function main: "<< x <<endl;
    return 0;
}
```

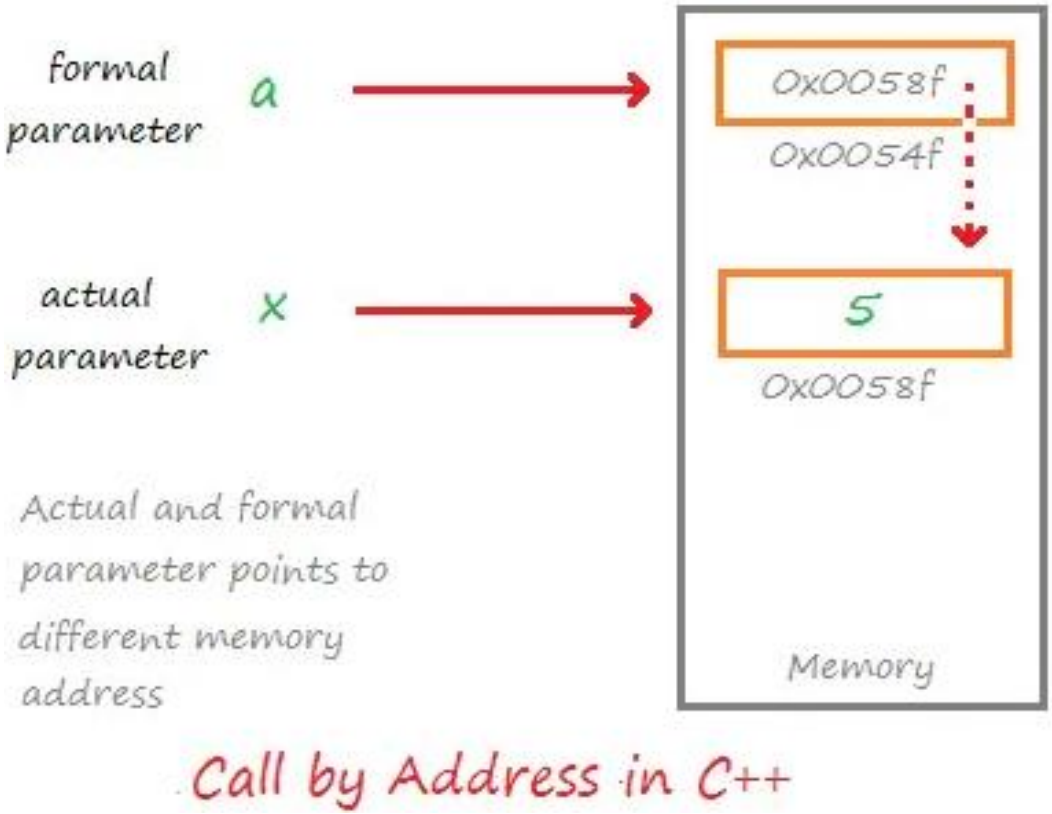
```
C:\Users\pc\Documents\Proje
Value in Function increment: 6
Value in Function main: 6

Process returned 0   execution time : 0.071 s
Press any key to continue.
```

User-defined Function: *Function Calling*

- **Call by Address:** In the call by address method, **both actual and formal parameters indirectly share the same variable.**
  - In this type of call mechanism, **pointer variables** are used as formal parameters.
  - The **formal pointer variable** holds the address of the actual parameter, hence the changes done by the formal parameter is also reflected in the actual parameter.

User-defined Function: *Function Calling*

- **Call by Address:** as demonstrated in the diagram, both parameters point to different locations in memory, but since the formal parameter stores the address of the actual parameter, they **share the same value**.



Call by Address in C++

User-defined Function: *Function Calling*

- Call by Address: Example-

```
#include <iostream>
using namespace std;

//a stores the address of x
void increment(int *a){
    (*a)++;
    cout << "Value in Function increment: "<< *a <<endl;
}

int main()
{
    int x = 5;
    increment(&x); //Passing address of x
    cout << "Value in Function main: "<< x <<endl;
    return 0;
}
```

```
C:\Users\pc\Documents\Proje
Value in Function increment: 6
Value in Function main: 6

Process returned 0   execution time : 0.070 s
Press any key to continue.
```

User-defined Function: *Function Calling*

- **Default Values for Parameters:**
  - When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.
  - If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead.

User-defined Function: *Function Calling*

```cpp
#include <iostream>
using namespace std;

int sum(int a, int b = 20) {
   int result;
   result = a + b;

   return (result);
}
int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;
   int result;

   // calling a function to add the values.
   result = sum(a, b);
   cout << "Total value is :" << result << endl;

   // calling a function again as follows.
   result = sum(a);
   cout << "Total value is :" << result << endl;

   return 0;
}
```

```
Total value is :300
Total value is :120
```

User-defined Function: *Function Calling*

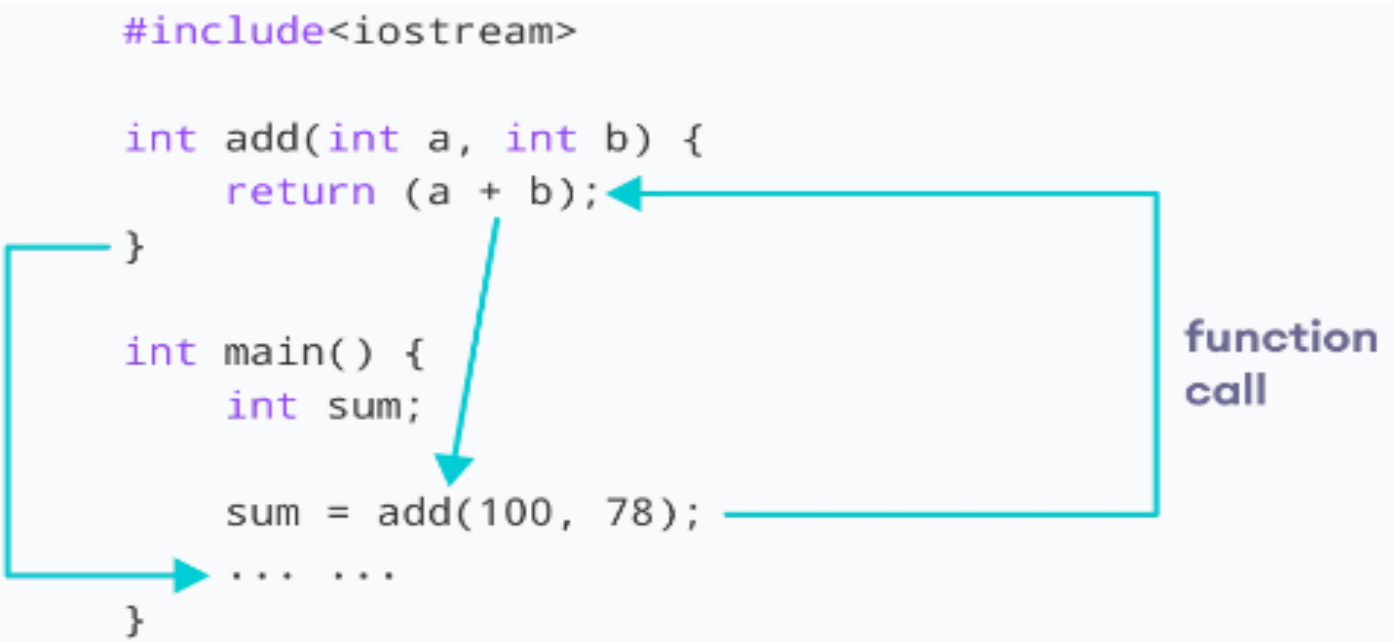
- **Constant Arguments:**
  - Just like we can declare constant variables, C++ also allows us to declare constant arguments which cannot be modified in the function.
  - For example:

    ```
    float example (int day, int month, const int year = 2005);
    ```

  - In this declaration the argument 'year' is declared as a constant variable.
  - The keyword const preceding this variable is an access specifier and it tells the compiler that the function should not modify the argument.

User-defined Function: *Function Calling*

- **Return Statement**
  - In the below program, the **add()** function is used to find the sum of two numbers.
  - We store the returned value of the function in the variable sum.
  - **Note**: sum is a variable of int type. This is because the return value of **add()** is of int type.



User-defined Function: *Function Calling*

- **Function Prototype**
  - In C++, the code of function declaration should be before the function call. However, if we want to define a function after the function call, we need to use the function prototype.

```
// function prototype
void add(int, int);

int main() {
    // calling the function before declaration.
    add(5, 3);
    return 0;
}

// function definition
void add(int a, int b) {
    cout << (a + b);
}
```

Benefits of User-defined Function

- Functions **make the code reusable**. We can declare them once and use them multiple times.
- Functions **make the program easier** as each small task is divided into a function.
- Functions **increase readability**.

C++ Library Functions

- Library functions are the built-in functions in C++ programming.
- Programmers can use library functions by invoking the functions directly; they don't need to write the functions themselves.
- Some common library functions in C++ are sqrt(), abs(), isdigit(), etc.
- In order to use library functions, we usually need to include the header file in which these library functions are defined.
- For instance, in order to use mathematical functions such as sqrt() and abs(), we need to include the header file **cmath**.

C++ Library Functions

- Example: C++ Program to Find the Square Root of a Number

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    double number, squareRoot;

    number = 25.0;

    // sqrt() is a library function to calculate the square root
    squareRoot = sqrt(number);

    cout << "Square root of " << number << " = " << squareRoot;

    return 0;
}
```

Function Overloading

- Function overloading is a feature of object-oriented programming where **two or more functions can have the same name** but **different parameters**.
- When a function name is overloaded with **different jobs** it is called Function Overloading.
- In Function Overloading "Function" name should be the same and the arguments should be different.
- Function overloading can be considered as an example of a **polymorphism** feature in C++.

Function Overloading

- Notice that the return types of all these 4 functions are not the same.

```
// same name different arguments
int test() { }
int test(int a) { }
float test(double a) { }
int test(int a, double b) { }
```

- Overloaded functions **may or may not have different return types** but they **must have different arguments**.
- For example:

```
// Error code
int test(int a) { }
double test(int b){ }
```

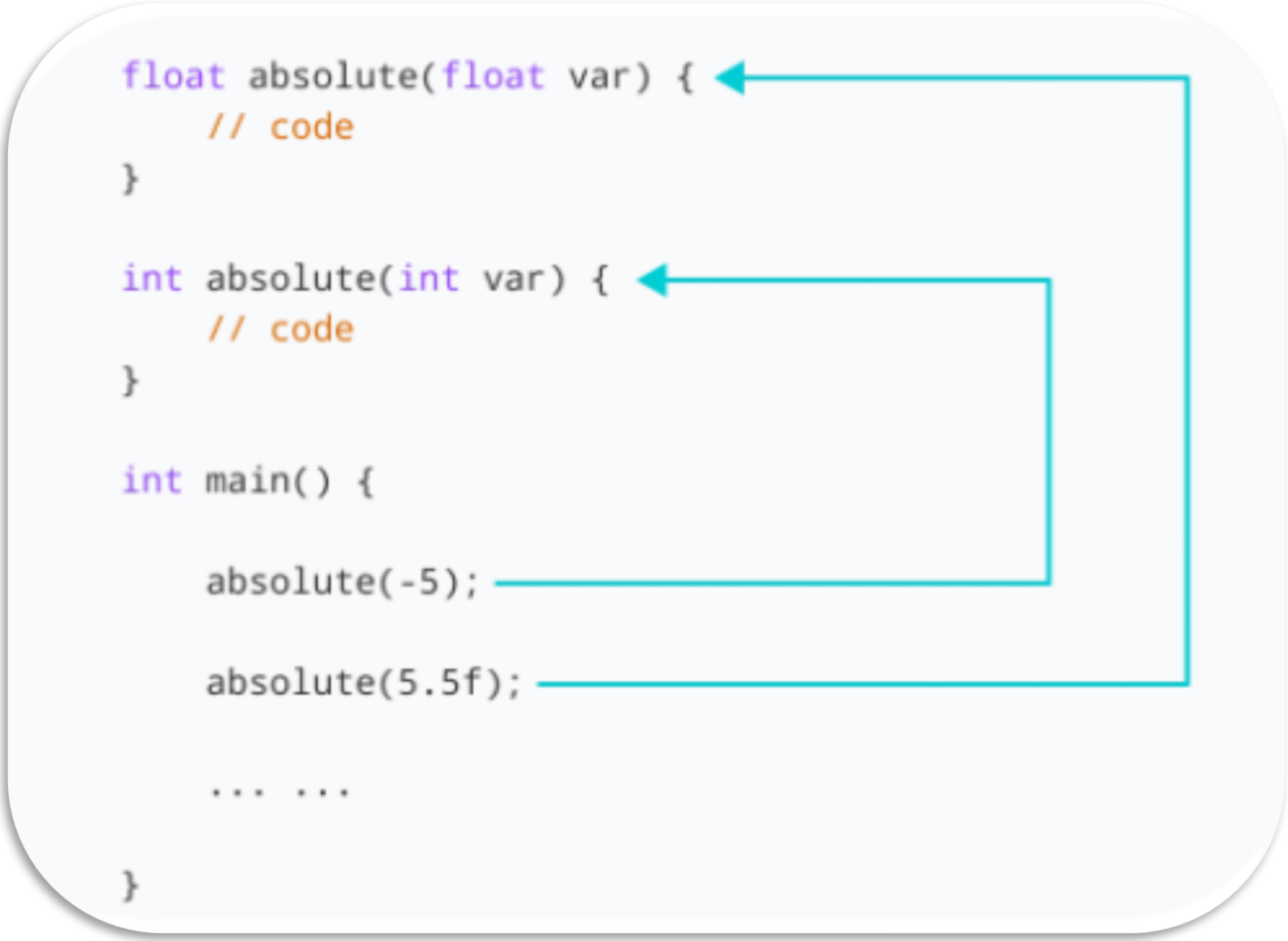
Function Overloading

```cpp
#include <iostream>
using namespace std;

// function with float type parameter
float absolute(float var){
    if (var < 0.0)
        var = -var;
    return var;
}

// function with int type parameter
int absolute(int var) {
     if (var < 0)
         var = -var;
    return var;
}

int main() {

    // call function with int type parameter
    cout << "Absolute value of -5 = " << absolute(-5) << endl;

    // call function with float type parameter
    cout << "Absolute value of 5.5 = " << absolute(5.5f) << endl;
    return 0;
}
```

Output

```
Absolute value of -5 = 5
Absolute value of 5.5 = 5.5
```

Function Overloading



Inline Functions

- C++ enable you to use inline functions that expand into their statements. Thus, inline functions offer faster execution time especially helpful where speed is critical at the cost of expanding the code.
- An inline function is a function that is expanded in line when it is called.
- When the inline function is called whole code of the inline function gets inserted or substituted at the point of the inline function call.
- This substitution is performed by the C++ compiler at compile time. .

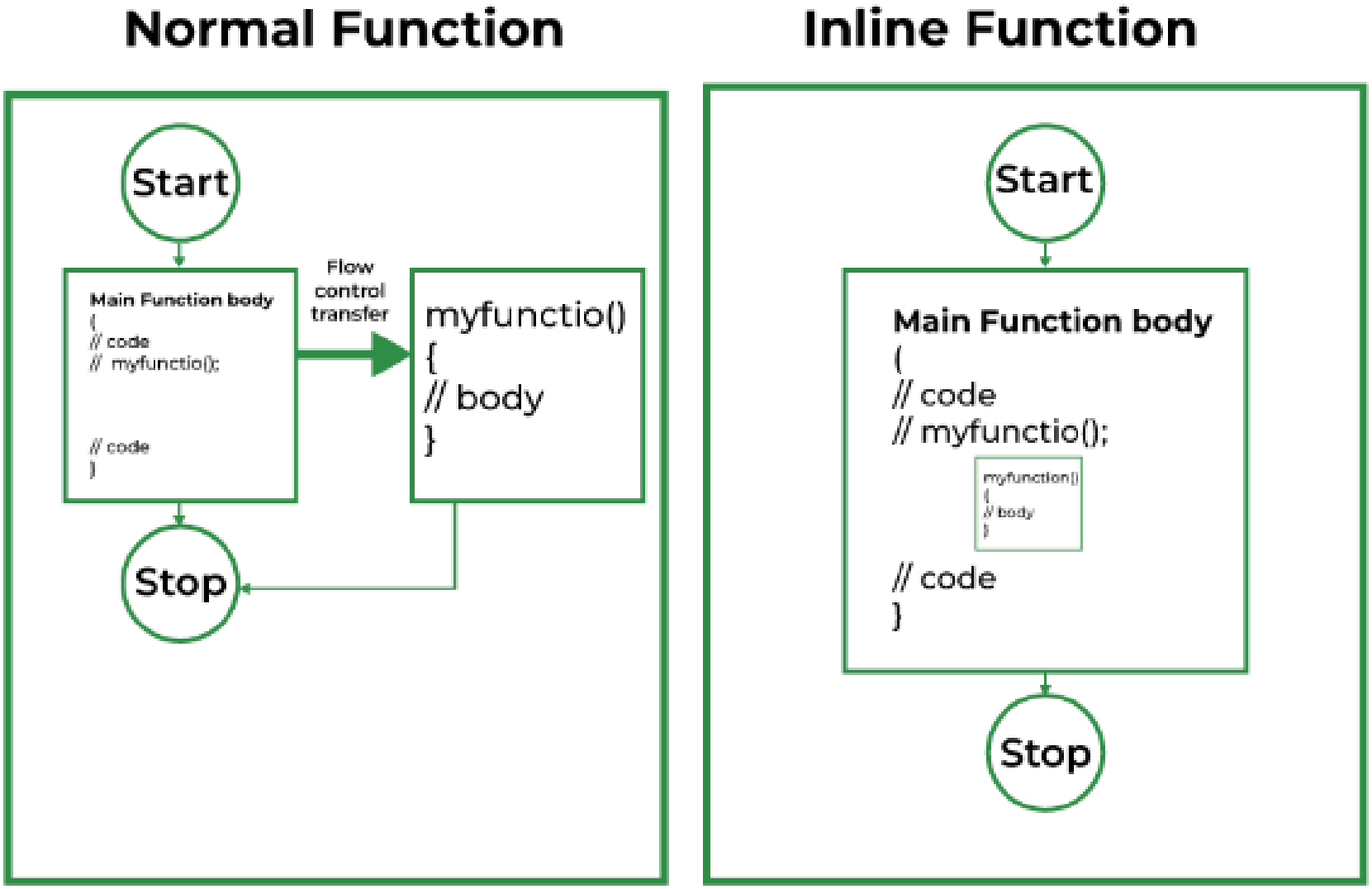
- The inline function can be defined as follows:

```
inline returnType functionName(DataType parameterList);
```

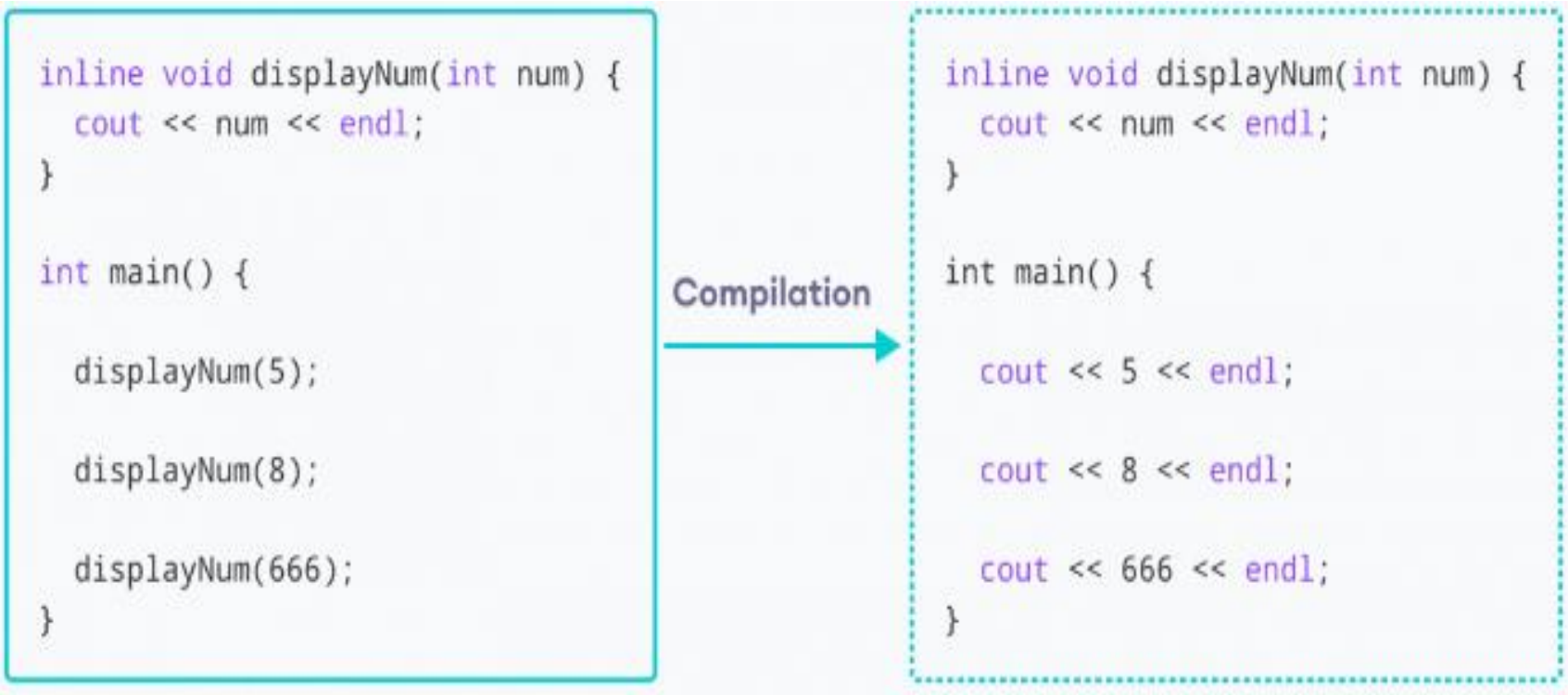
Inline Functions

- The compiler may not perform in lining in such circumstances as:
  - If a function contains a loop. (for, while and do-while)
  - If a function contains static variables.
  - If a function is recursive.
  - If a function return type is other than void, and the return statement doesn't exist in a function body.
  - If a function contains a switch or goto statement.

Inline Functions



Inline Functions



Recursive functions

- When function is called within the same function, it is known as **recursion** in C++. The function which calls the same function, is known as **recursive function**.

- A function that calls itself, and doesn't perform any task after function call, is known as **tail recursion**. In tail recursion, we generally call the same function with return statement.

- A function that calls itself is known as a **recursive function**. And, this technique is known as **recursion**.

Recursive functions: Types of Recursion

1. **Direct recursion:** In direct recursion, the function calls itself directly.

```
#include <iostream>
using namespace std;

void direct(){
    //Statements
    direct()
}
int main(){

    return 0;
}
```

Recursive functions: Types of Recursion

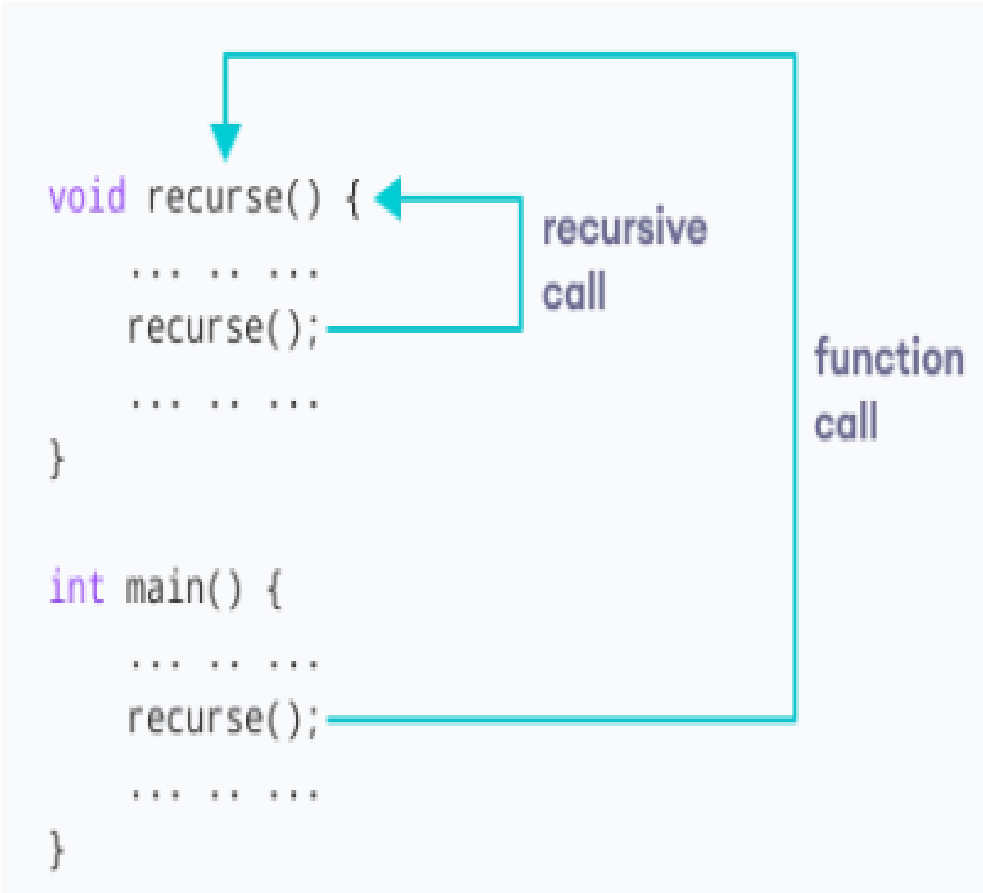
2. **Indirect recursion:** If a function calls itself indirectly from another function, then this type of recursion is called indirect recursion.

```
#include <iostream>
using namespace std;

void indirect(){
    //Statements
    func()
}
void func(){
    //Statements
    indirect()
}

int main(){

    return 0;
}
```

Recursive functions

- Working of Recursion in C++

```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```



Recursive functions

- Example 1: Factorial of a Number Using Recursion

```
// Factorial of n = 1*2*3*...*n

#include <iostream>
using namespace std;

int factorial(int);

int main() {
    int n, result;

    cout << "Enter a non-negative number: ";
    cin >> n;

    result = factorial(n);
    cout << "Factorial of " << n << " = " << result;
    return 0;
}

int factorial(int n) {
    if (n > 1) {
        return n * factorial(n - 1);
    } else {
        return 1;
    }
}
```

Output

```
Enter a non-negative number: 4
Factorial of 4 = 24
```

Recursive functions

- Example 2: find the sum of all the digits up to a particular number, i.e. entered by the user with the help of recursion.

```
#include <iostream>
using namespace std;

int sum(int num)
{
    if (num != 0)
    {
        return num + sum(num - 1);
    }
    return 0;
}

int main()
{
    int num;
    cout << "Enter the number : ";
    cin >> num;
    cout << "Sum is : " << sum(num);

    return 0;
}
```

Exercises

1. Write a program to take two numbers say, 'x' and 'y' from the user and add these numbers using a function and then display the sum of these two numbers on the screen.
2. Write a program which declares a function '**int accept(int a, int b, int c);**' which accepts three parameters 'a', 'b' and 'c' inputted by the user and returns the value obtained by computing value of the expression '**a * b +c**' to the main program which displays the returned value.
3. Write a program to find the roots of a quadratic equation. Assume that the quadratic equation only has unequal or equal roots. The program should use separate set of functions to determine whether the equation has equal or unequal roots.

Exercises

4. Write a program to display the following type of pattern using functions (The value of n entered by the user)?

```
        *
      * *
    * * *
  * * * *
* * * * *
```

5. Write a program which declares a function which prints the Fibonacci series on the screen, where the number of terms to be displayed is passed as a parameter by the user.

6. Declare a function 'multiply' which takes in two integers as arguments and returns their product as an integer.

Chapter Five

Arrays, Strings, and Pointer

Part 1

Arrays in C++

Outlines

- Arrays in C++
  - What does it mean?
  - Declaration of an Array
  - Accessing Elements of an Arrays
  - Input/Output of an Array
  - Searching a Value in an Array
  - Address of an Array
  - Multidimensional arrays
    - Two Dimensional Arrays
    - Three Dimensional Arrays
  - Passing an Array to a Functions

Overview of C++ Arrays

- An array is a **collection of identical data objects** (i.e. can only hold values of one type), which are stored in consecutive memory locations under a common variable name.

- It's a group of values referred to by the same name.

- The objects (i.e. individual values in array) are called the elements of the array and are numbered consecutively from **0** up to **n-1**.

Overview of C++ Arrays

- Advantages of C++ Array
  - Code Optimization (less code)
  - Random Access
  - Easy to traverse data
  - Easy to manipulate data
  - Easy to sort data etc.
- Disadvantages of C++ Array
  - Fixed size
- C++ Array Types
  - Single Dimensional Array- **1D**
  - Multidimensional Array- **2D, 3D, … , ND**

Declaration of an Array

- Declaring/defining the name and type of an array and setting the number of elements in an array is called *dimensioning the array*.
  - The array must be declared before one uses in like other variables.
- **Syntax:** *dataType* **arrayName** [**arraySize**] ;
  - Example: *double* **balance**[5];
  - *dataType* - the data type of the elements of the array (e.g. int, float, char, double, etc.)
  - **arrayName** - the name or identifier of the array,
  - **arraySize** - the total number of memory locations to be allocated or the maximum value of each subscript. i.e. the total number of elements in the array.

Initialization of an Array

- You can initialize C++ array elements either one by one or using a single statement as follows

```
double balance[5] = {10.0, 2.0, 3.4, 17.0, 50.0};
double balance[ ] = {1000.0, 2.0, 3.4, 17.5, 50.6};
```

Array Indices	0	1	2	3	4
Array Data	10.0	2.0	3.4	17.5	50.6

- The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ].

Address of an array

- The values of any of the elements in an array can be accessed just like the value of a regular variable of the same type.
- **Syntax:** arrayName[index];

- So, from the previous example the elements of the array **balance** are as follows:
  - balance[0] = 10.0
  - balance[1] = 2.0
  - balance[2] = 3.4
  - balance[3] = 17.5
  - balance[4] = 50.6

Accessing Elements of an Array

```
#include <iostream>
using namespace std;
int main () {
  int n[ 10 ]; // n is an array of 10 integers

// initialize elements of array n to 0
  for ( int i = 0; i < 10; i++ ) {
    n[ i ] = i + 100; // set element at location i to i
+ 100
  }
  cout << "Element\t\t\tValue" << endl;

// output each array element's value
  for ( int j = 0; j < 10; j++ ) {
    cout <<"\t" << j << "\t\t\t" << n[ j ] << endl;
  }
 return 0;
}
```

Element	Value
0	100
1	101
2	102
3	103
4	104
5	105
6	106
7	107
8	108
9	109

Address of an array

- With the declaration of array as **int section[3];** three blocks of memory are allocated by the compiler for the array elements, i.e. one block for each element for storing its value. The array is initialized as given below.

int section [3] = {1,2,3};

- The size of blocks depends on the type of data in the array. For the present case, the type of elements is int so 2 bytes are allocated for each block. The value of each element is stored in binary as shown below.

Array elements	0	1	2	
	0001	0010	0011	Values of array elements
Memory address	AF000000	AF000001	AF000010	

Address of an array

- The address of an array is the **address of first element of the array** which is the number of the first byte of the memory block allocated to the first element of the array.

  - It may be extracted by **simply calling the name of array**. Thus, for array section [3] the code for getting address of the array is:

```
cout << section; //returns the address of an array section/address of section [0]
```

Address of an array

- The output will be the address of the array. The address of any element of an array may also be extracted by calling array (**name + offset**).
  - The **offset** is equal to **the subscript or index value** of the element.

cout << section+1; *//returns the address of the second element (i.e. section [2]).*

cout << section+2; *//returns the address of the third element (i.e. section [3]).*

Multidimensional Arrays

- If the elements of an array are arrays, the array is a multi-dimensional array. Thus, an array having elements which are one dimensional arrays, is a two-dimensional array. Similarly, an array whose elements are two-dimension arrays is a three-dimensional array and so forth for others.
- Let the 5 elements of an array **Names** be arrays of 30 characters each.

char Names [5][30];

Multidimensional Arrays

- Two-dimensional array example:

```
int M[2][5]= {5,2,3,2,4,6,7,8,9,8};or
    int M[2][5]= {
          {5,2,3,2,4}, /* initializers for row indexed by 0 */
          {6,7,8,9,8} /* initializers for row indexed by 1 */
          };
```

- In computer memory a two-dimensional array is also stored in the same way as shown below. i.e. two rows and 5 columns like a matrix.

M [2][5]				
5	2	3	2	4
6	7	8	9	8

Multidimensional Arrays

```
#include <iostream>
using namespace std;
int main () {
   // an array with 5 rows and 2 columns.
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
   // output each array element's value
   for ( int i = 0; i < 5; i++ )
      for ( int j = 0; j < 2; j++ ) {
         cout << "a[" << i << "][" << j << "]: ";
         cout << a[i][j]<< endl;
      }
   return 0;
}
```

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

Passing arrays as an argument

- C++ **does not allow to pass an entire array** as an argument to a function. However, you can pass a **pointer to an array** by specifying the **array's name without an index.**
- If you want to pass a single-dimension array as an argument in a function, you would have to declare function formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received.

Passing arrays as an argument

- **Way 1:** Formal parameters as a pointer as follows:

```
void myFunction(int *param){
    // Statements here
}
```

- **Way 2:** Formal parameters as a sized array as follows:

```
void myFunction(int param[10]){
    // Statements here
}
```

- **Way 3:** Formal parameters as an unsized array as follows:

```
void myFunction(int param[]){
    // Statements here
}
```

Passing arrays as an argument

- By passing a particular element of the array to the function

```cpp
#include <iostream>
using namespace std;

void pass(int arr)
{
   cout<<"The element is "<<arr;
}

int main()
{
    int arr[5]={1,3,9,4,5};

    pass(arr[3]);
    return 0;
}
```

```
C:\Users\pc\Documents\Proje
The element is 4
Process returned 0   execution time : 0.164 s
Press any key to continue.
```

Passing arrays as an argument

- By passing the whole array to the function:

```
#include <iostream>
using namespace std;

void pass(int arr[])
{
    for(int i=0;i<5;i++)
    {
      cout<<"The element is "<<arr[i]<<endl;
    }
}

int main()
{
    int arr[5]={1,3,9,4,5};

    pass(arr);
    return 0;
}
```

```
C:\Users\pc\Documents\Proje
The element is 1
The element is 3
The element is 9
The element is 4
The element is 5

Process returned 0   execution time : 0.060 s
Press any key to continue.
```

Passing arrays as an argument

- When a two-dimensional array is used as an argument to a function, only a **pointer to the first element** is actually passed.
- However, the parameter receiving a two-dimensional array must define at least the size of the rightmost dimension. (You can specify the left dimension if you like, but it is not necessary.)

https://www.scaler.com/topics/cpp/strings-in-cpp/

Part 2

C++ String

Outlines

- Strings in C++
  - Overview
  - The C-style character string
  - String vs. Character Array
  - Concatenation of Strings
  - Passing Strings to Functions
  - C++ Built-in String Functions
  - Operations of C++ Strings
    - Input functions
    - Capacity Functions
    - Iterator Functions
    - Manipulating Functions

Overview of C++ Strings

- A string in C++ is a type of object representing a collection (or sequence) of different characters surrounded by double quotes.
- Strings in C++ are a part of the standard string class (std::string).
- The string class stores the characters of a string as a collection of bytes in contiguous memory locations.
- In order to use the string data type, the C++ string header <string> must be included at the top of the program.
- C++ provides following two types of string representations-
  - The C-style character string.
  - The string class type introduced with Standard C++.

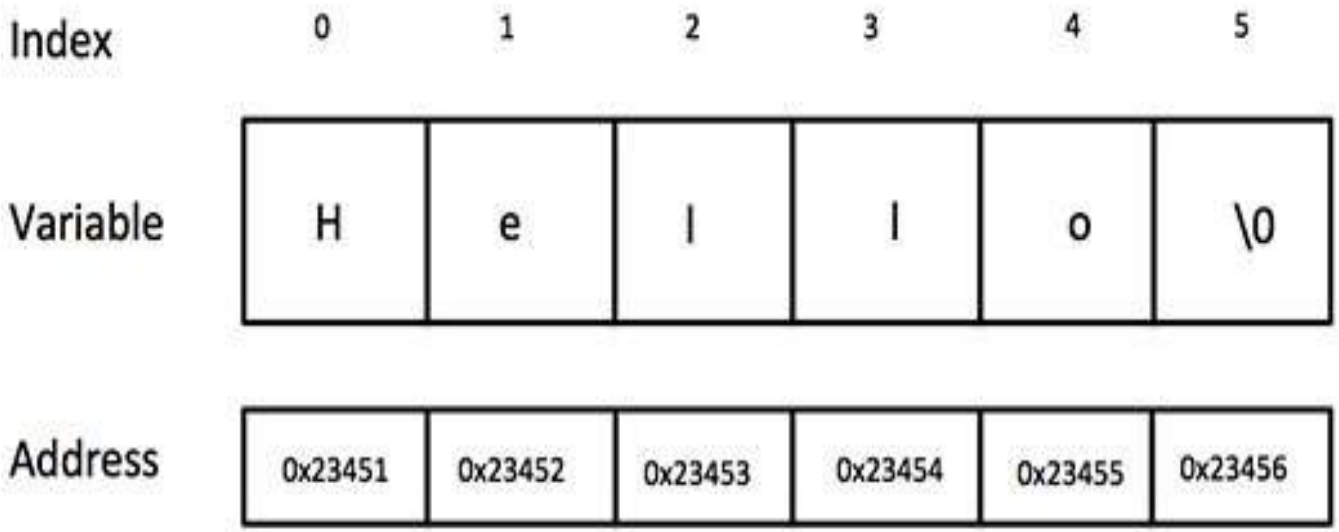
The C-style character string

- The C-style character string originated within the C language and continues to be supported within C++.
- This string is actually a one-dimensional array of characters which is **terminated by a null character '\0'**.
- Thus a null-terminated string contains the characters that comprise the string followed by a null.
- The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is **one more** than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
char greeting[] = "Hello";
```

The C-style character string

- Following is the memory presentation of above defined string in C/C++.



- Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array.

The C-style character string

- Here are the different ways to create C-style strings:

```
char str_name[6] = {'h', 'e', 'l', 'l', 'o', '\0'};

// or

char str_name[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

- Alternatively, we have:

```
char str_name[] = "hello";

// or

char str_name[6] = "hello";
```

- It is not necessary to utilize all the space allocated to a character array (string):

```
char str_name[50] = "hello";
```

The C-style character string

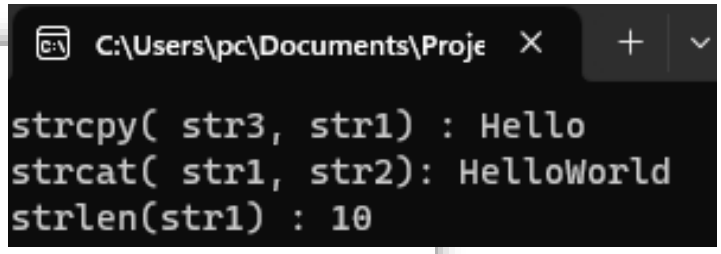
- C++ supports a wide range of functions that manipulate null-terminated strings-

No	Function	Purpose
1	**strcpy(s1, s2);**	Copies string s2 into string s1.
2	**strcat(s1, s2);**	Concatenates string s2 onto the end of string s1.
3	**strlen(s1);**	Returns the length of string s1.
4	**strcmp(s1, s2);**	Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);**	Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);**	Returns a pointer to the first occurrence of string s2 in string s1.

The C-style character string

- *Example*

```
#include <iostream>
#include <cstring>
using namespace std;

int main () {
   char str1[10] = "Hello", str2[10] = "World";
   char str3[10];
   int  len ;

   // copy str1 into str3
   strcpy( str3, str1);
   cout << "strcpy( str3, str1) : " << str3 << endl;

   // concatenates str1 and str2
   strcat( str1, str2);
   cout << "strcat( str1, str2): " << str1 << endl;

   // total lenghth of str1 after concatenation
   len = strlen(str1);
   cout << "strlen(str1) : " << len << endl;

   return 0;
}
```



The String Class in C++

- The standard C++ library provides a **string** class type that supports all the operations mentioned above, additionally much more functionality.
- Using the **string keyword** is the best and the most convenient way of defining a string.
- We should prefer using the string keyword because it is **easy to write and understand**.

```
string str_name = "hello";

// or

string str_name("hello");
```

String vs. Character Array

Comparison	Strings	Character Array
Definition	String is a C++ class while the string variables are the objects of this class	Character array is a collection of variables with the data type char.
Syntax	string string_name;	char arrayname[array_size];
Access Speed	Slow	Fast
Indexing	To access a particular character, we use "str_name.charAt(index)" or "str[index]".	A character can be accessed by its index in the character array.

String vs. Character Array

Comparison	Strings	Character Array
Operators	Standard C++ operators can be applied.	Standard C++ Operators can not be applied.
Memory Allocation	Memory is allocated dynamically. More memory can be allocated at run time.	Memory is allocated statically. More memory can not be allocated at run time.
Array Decay	Array decay (loss of the type and dimensions of an array) is impossible.	Array decay might occur.

Concatenation of Strings

- Combining two or more strings to form a resultant string is called the concatenation of strings.
- In C++, we have three ways to concatenate strings. These are as follows.
  1. Using **strcat()** function
  2. Using **append()** function
  3. Using the '**+**' operator

Concatenation of Strings

- Using strcat() function
  - We need to include the cstring header file in our program.
  - The strcat() function takes two character arrays as the input. It concatenates the second array to the end of the first array.
  - Syntax: strcat(char_array1, char_array2);
  - It is important to note that the strcat() function only accepts character arrays as its arguments.
    - We can not use string objects in the function.

Concatenation of Strings

- Using strcat() function- Example

```
#include <iostream>
#include <cstring>
using namespace std;

int main() {
  char str1[] = "Scaler ";
  char str2[] = "Topics.";

  strcat(str1, str2);

  cout << str1 << endl;

  return 0;
}
```

The output of the above program is:

```
Scaler Topics.
```

Concatenation of Strings

- Using append() function
  - The append() function appends the first string to the second string.
  - The variables of type string are used with this function.
  - Syntax: str1.append(str2);

Concatenation of Strings

- Using append() function- Example

```cpp
#include <iostream>
using namespace std;

int main() {
   string str1 = "Scaler ";
   string str2 = "Topics.";

   str1.append(str2);

   cout << str1 << endl;

   return 0;
}
```

Output:

```
Scaler Topics.
```

Concatenation of Strings

- Using the '+' operator
  - This is the easiest way to concatenate strings.
  - The + operator takes two (or more) strings and returns the concatenated string.
  - Syntax: str1 + str2;

- It is up to the programmer to use + or **append()**. Compared to +, the **append()** function is much faster. However, in programs that do not have very long strings, replacing + with **append()** will not significantly affect program efficiency. On the other hand, the **strcat()** function has some potential performance issues based on the function's time complexity and the number of times the function runs in the program. So, in a nutshell, it is best to use the **append()** function or the + operator.

Concatenation of Strings

- Using the '+' operator- Example

```cpp
#include <iostream>
using namespace std;

int main() {
  string str1 = "Scaler ";
  string str2 = "Topics.";

  str1 = str1 + str2;

  cout << str1 << endl;

  return 0;
}
```

Output:

```
Scaler Topics.
```

Passing Strings to Functions

- This is the same as we do with other arrays. The only difference is that this is an array of characters.

```
#include <iostream>
using namespace std;

void display( char ch[] ){
    cout << ch;
}

int main(){
    char arr[30];
    cout << "Enter a word: ";
    cin >> arr;
    display(arr);
    return 0;
}
```

```
C:\Users\pc\Documents\Proje
Enter a word: Computer Programming
Computer
Process returned 0   execution time : 9.122 s
Press any key to continue.
```

C++ Built-in String Functions

Functions	Description
int length()	Used to find the string length
void swap(string& str1)	Used to swap the values of 2 strings
int compare(const string& str1)	Used to compare 2 strings
string substr(int position, int length)	Used to create a new string of a given length
int size()	Used to find the string length in terms of bytes

C++ Built-in String Functions

Functions	Description
void resize(int length)	Used to resize the string length up to the given no. of characters
string& replace(int pos, int n, string& str)	Used to replace a portion of the string that begins at position pos and is of the length of n characters
string& append(const string& str)	Used to add new characters at the end of a string
char& at(int position)	Used to access an individual character at the given position

C++ Built-in String Functions

Functions	Description
int find(string& str1, int position, int len)	Used to find the string given in the parameter
int find_first_of(string& str1, int position, int len)	Used to find the first occurrence of the given sequence
int find_first_not_of(string& str1, int position, int n)	Used to search the string for the first character that does not match any characters specified in the string
int find_last_of(string& str1, int position, int n)	Used to search the string for the last character of the sequence specified

C++ Built-in String Functions

Functions	Description
int find_last_not_of(string& str1, int position)	Used to search for the last character that does not match with the sequence specified
string& insert()	Used to insert a new character before the character indicated at the given position
int max_size()	Used to get the maximum length of a string
void push_back(char c)	Used to add a new character at the end of a string

C++ Built-in String Functions

Functions	Description
void pop_back()	Used to remove the string's last character
string& assign()	Used to assign new value to a string
int copy(string& str1)	Used to copy the contents of a string into another
begin()	Used to return the first character's reference
char& back()	Used to return the last character's reference
int capacity()	Used to return the allocated space for the string

C++ Built-in String Functions

Functions	Description
void clear()	Used to remove string's all elements
const_iterator cbegin()	Used to point to the string's first element
const_iterator cend()	Used to point to the string's last element
const_reverse_iterator crbegin()	Used to point to the string's last character
bool empty()	Used to check if the string is empty or not
string& erase()	Used to remove the characters
const_char* data()	Used to copy the string's characters into an array

C++ Built-in String Functions

Functions	Description
char& front()	Used to return a reference of the string's first character
string& operator=()	Used to assign a new value to a string
string& operator+=()	Used to insert a new character at the end of a string
char operator[](position)	Used to retrieve a character at the position specified
int rfind()	Used to search for the string's last occurrence
iterator end()	Used to reference the string's last character

C++ Built-in String Functions

Functions	Description
reverse_iterator rend()	Used to point to the string's first character
void shrink_to_fit()	Used to reduce the capacity of the container and make it equal to the size of the string
char* c_str()	Used to return pointer to an array containing null-terminated character sequence
const_reverse_iterator crend()	Used to reference the first character of the string

C++ Built-in String Functions

Functions	Description
allocator_type get_allocator();	Used to return the allocated object connected with a string
reverse_iterator rbegin()	Used to reference the last character of a string
void reserve(int length)	Used to reserve the vector capacity

Operations of C++ Strings

- Although the implementation of **character arrays is faster than that of string objects**, string objects are widely used in C++ because of the number of in-built functions it supports.

- Some of the most commonly used string functions are as follows:
  - Input functions
  - Capacity Functions
  - Iterator Functions
  - Manipulating Functions

C++ String Input Functions

- The input functions help add or remove a character or string to a string.

Functions	Description
getline()	Used to read and store a string that the user enters through the input stream
push_back()	Inputs a new character to the string's end
pop_back()	Pops out (deletes) a string's last character (introduced from C++11)

C++ String Input Functions

- **getline():** it is used to read a string entered by the user.
  - The getline() function extracts characters from the input stream.
  - It adds it to the string object until it reaches the delimiting character.
  - The default delimiter is **\n**.
    - The delimiter could **either be a character or the number of characters a user can input**.

C++ String Input Functions

- getline()- Example

```cpp
#include <iostream>
using namespace std;

int main() {
  char profession[12];

  cout << "Enter your profession: ";
  // We will take a maximum of 12 characters as input
  cin.getline(profession, 12);

  cout << "Your profession is: " << profession;

  return 0;
}
```

```
C:\Users\pc\Documents\Proje
Enter your profession: Computer Science
Your profession is: Computer Sc
Process returned 0   execution time : 9.319 s
Press any key to continue.
```

C++ String Input Functions

- **push_back():** it is used to push (add) characters at the end (back) of a string.
  - After the character is added, the size of the string increases by 1.
  - For example using the push_back() function several times, we added the word " Students" at the end of the string.

C++ String Input Functions

push_back()- Example

```cpp
#include <iostream>
using namespace std;

int main() {
  string name = "Computer Science";
  cout << "Name before using push_back: " << name << endl;

  // Let's add the word "Topics" at the end of the above string
  name.push_back(' ');
  name.push_back('S');
  name.push_back('t');
  name.push_back('u');
  name.push_back('d');
  name.push_back('e');
  name.push_back('n');
  name.push_back('t');
  name.push_back('s');
  cout << "Name after using push_back: " << name << endl;
  return 0;
}
```

```
C:\Users\pc\Documents\Proje
Name before using push_back: Computer Science
Name after using push_back: Computer Science Students

Process returned 0   execution time : 0.041 s
Press any key to continue.
```

C++ String Input Functions

- **pop_back():** It is used to pop (remove) 1 character from the end (back) of a string.
  - As the character gets removed, the size of the string decreases by 1.
  - As we can see from the following example, the last character of the name popped out, and the string is left with one less character.

C++ String Input Functions

pop_back()- Example

```cpp
#include <iostream>
using namespace std;

int main() {
  string name = "Scaler";

  cout << "Name before using pop_back: " << name << endl;

  // using the pop_back() function
  name.pop_back();

  cout << "Name after using pop_back: " << name << endl;
  return 0;
}
```

Output:

```
Name before using pop_back: Scaler
Name after using pop_back: Scale
```

C++ String Capacity Functions

- The C++ string capacity functions help to work with the size and capacity of the string. The primary functions of the capacity type are:

Functions	Description
capacity()	Returns the capacity allocated to a string by the compiler
resize()	Allows increasing or decreasing the size of the string
length()	Returns the size of the string
shrink_to_fit()	Decreases and makes the capacity equal to the minimum to save memory

C++ String Capacity Functions

- **length():** As the name suggests, the length() function returns the length of a string.
- **Example-**

```cpp
#include <iostream>
using namespace std;

int main() {
  string str = "Calculating length of the string";

  cout << "The length of string is: " << str.length() << endl;

  return 0;
}
```

Output:

```
The length of string is: 28
```

C++ String Capacity Functions

- **capacity():** It returns the current size of the allocated space for the string. The size can be equal to or greater than the size of the string. We allocate more space than the string size to accommodate new characters in the string efficiently.

```cpp
#include <iostream>
using namespace std;

int main() {
  string str = "C++ Programming";

  cout << "The length of string is: " << str.length() << endl;
  cout << "The capacity of string is: " << str.capacity() << endl;

  return 0;
}
```

Output:

```
The length of string is: 15
The capacity of string is: 15
```

C++ String Capacity Functions

- **resize():** It is used to either increase or decrease the size of a string.
  - In the below example, we used the resize() function to change the number of characters in the string. Because we had specified the first ten characters in resize(), all the characters after the first 10 were removed from the string.

```cpp
#include <iostream>
using namespace std;

int main() {
  string str = "C++ Programming";
  cout << "The original string is: " << str << endl;

  // We will keep only the first ten characters of the string
  str.resize(10);
  cout << "The string after using resize is: " << str << endl;
  return 0;
}
```

```
C:\Users\pc\Documents\Proje
The original string is: C++ Programming
The string after using resize is: C++ Progra

Process returned 0   execution time : 0.226 s
Press any key to continue.
```

C++ String Capacity Functions

- **shrink_to_fit():** It is used to decrease the string's capacity to make it equal to the string's minimum capacity. This function helps us save memory if we are sure that no additional characters are required.

  - In the above example, after using the resize() function, we realized that the string capacity was 28. When we used the shrink_to_fit() function, the size was reduced to 17.

C++ String Capacity Functions

- shrink_to_fit():

```
#include <iostream>
using namespace std;

int main() {
    string str = "Using shrink to fit function";

    // using resize
    str.resize(17);

    cout << "The capacity of string before using shrink_to_fit is: " <<
    str.capacity() << endl;

    str.shrink_to_fit();

    cout << "The capacity of string after using shrink_to_fit is: " <<
    str.capacity() << endl;

    return 0;
}
```

Output:

```
The capacity of string before using shrink_to_fit is: 28
The capacity of string after using shrink_to_fit is: 17
```

C++ String Iterator Functions

- The iterator functions are used to work with the iterators that traverse through each character of a string.

Functions	Description
begin()	Returns the iterator to the beginning
end()	Returns the iterator to the end
rbegin()	Returns a reverse iterator that points to the end
rend()	Returns a reverse iterator that points to the start

C++ String Iterator Functions

- **begin():** It returns an iterator that points to the beginning of the string.

```
#include <iostream>
using namespace std;

int main() {
    string str = "Learning C++";

    // declaring an iterator
    string::iterator it;

    it = str.begin();

    cout << * it << " ";
    it++;

    cout << * it;
    return 0;
}
```

```
C:\Users\pc\Documents\Proje
L e
Process returned 0   execution time : 0.336 s
Press any key to continue.
```

C++ String Iterator Functions

- **end():** This function returns an iterator that points to the end of the string. The iterator will always point to the null character.

```cpp
#include <iostream>
using namespace std;
int main() {
  string str = "Programming";
  // declaring an iterator
  string::iterator it;
  it = str.end();

  // nothing will print because the iterator points to \0
  cout << * it;
  it--;

  cout << * it;
  it--;
  cout << * it;
  return 0;
}
```

```
C:\Users\pc\Documents\Proje
gn
Process returned 0   execution time : 0.080 s
Press any key to continue.
```

C++ String Iterator Functions

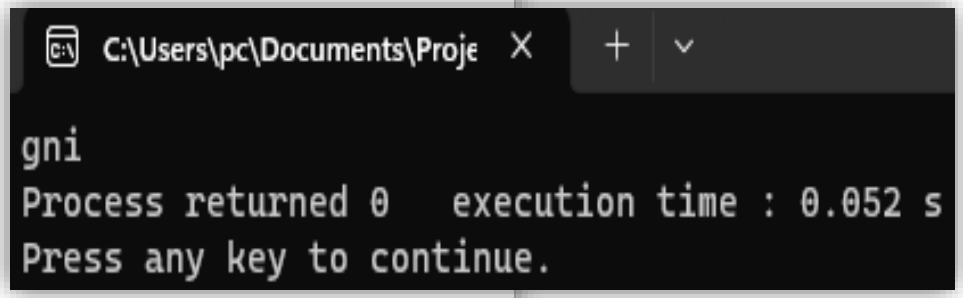
- **rbegin():** The keyword rbegin stands for the reverse beginning. It is used to point to the last character of the string.

  - The difference between **rbegin()** and **end()** is that end() points to the element next to the last element of the string while rbegin() points to the last element of a string.

C++ String Iterator Functions

- rbegin(): Example-

```cpp
#include <iostream>
using namespace std;

int main() {
  string str = "Programming";

  // declaring a reverse iterator
  string::reverse_iterator rit;
  rit = str.rbegin();

  cout << * rit;
  rit++;

  cout << * rit;
  rit++;

  cout << * rit;
  return 0;
}
```



C++ String Iterator Functions

- *rend(): The keyword rend stands for the reverse ending. It is used to point to the first character of the string.*

```
#include <iostream>
using namespace std;
int main() {
  string str = "Learning C++";
  // declaring a reverse iterator
  string::reverse_iterator rit;

  rit = str.rend();
  rit--;
  cout << * rit;
  rit--;
  cout << * rit;
  rit--;
  cout << * rit;
  return 0;
}
```

```
C:\Users\pc\Documents\Proje
Lea
Process returned 0   execution time : 0.061 s
Press any key to continue.
```

C++ String Manipulating Functions

- The manipulating functions allow you to manipulate a string with operations such as copying and swapping.

Functions	Description
copy("char array", len, pos)	It copies a substring from the char array and copies it. • The len and pos parameters define the length of the substring to be copied and starting position from where the copies begin, respectively.
swap()	Swaps one string with another

C++ String Manipulating Functions

- **copy():** It is used to copy one sub-string to the other string (character array) mentioned in the function's arguments.

  - It takes three arguments (minimum two)-
    - target character array,
    - length to be copied, and
    - starting position in the string to start copying.

C++ String Manipulating Functions

- copy(): Example-

```cpp
#include <iostream>
using namespace std;

int main() {
  string str1 = "Using copy function";
  char str2[10];

  // copying nine chars starting from index 5
  str1.copy(str2, 9, 5);

  // The last character of a char array should be \0
  str2[9] = '\0';

  cout << "The copied string is: " << str2;

  return 0;
}
```

```
C:\Users\pc\Documents\Proje
The copied string is:  copy fun
Process returned 0   execution time : 0.069 s
Press any key to continue.
```

C++ String Manipulating Functions

- **swap():** It swaps one string with another.

```cpp
#include <iostream>
using namespace std;

int main() {
  string fruit1 = "Apple";
  string fruit2 = "Orange";

  cout << "Before swap - fruit 1 is: " << fruit1 << endl;
  cout << "Before swap - fruit 2 is: " << fruit2 << endl << endl;

  // swapping
  fruit1.swap(fruit2);

  cout << "After swap - fruit 1 is: " << fruit1 << endl;
  cout << "After swap - fruit 2 is: " << fruit2 << endl;

  return 0;
}
```

```
C:\Users\pc\Documents\Proje
Before swap - fruit 1 is: Apple
Before swap - fruit 2 is: Orange

After swap - fruit 1 is: Orange
After swap - fruit 2 is: Apple

Process returned 0   execution time : 0.059 s
Press any key to continue.
```

1. https://www.scaler.com/topics/cpp/pointers-in-cpp/
2. https://www.geeksforgeeks.org/cpp-pointers/
3. https://www.programiz.com/cpp-programming/pointers
4. https://www.tutorialspoint.com/cplusplus/cpp_pointers.htm

Part 3

C++ Pointers

Outlines

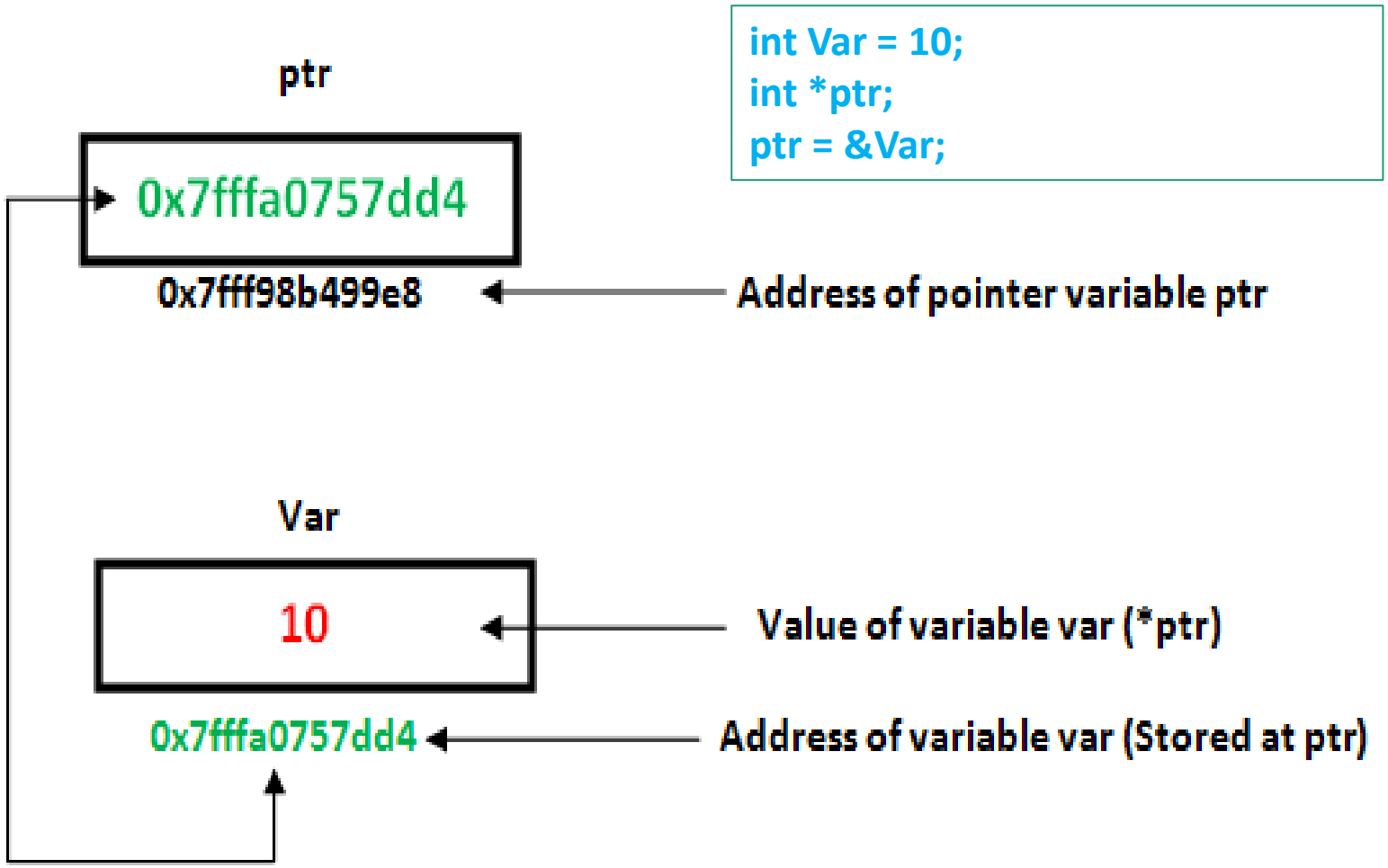
- Pointers in C++
  - Overview
  - Declaring pointers
  - Assigning values to pointers
  - Pointer to void
  - Invalid pointers
  - Null pointers
  - Arrays of pointers
  - Pointers and arrays
  - Pointer arithmetic
  - Pointer and string
  - Pointer to pointer
  - *Dynamic memory*

Overview

- A pointer in C++ is a variable that **stores the address** (or memory location) of another variable.
- In other words, a pointer **points to the address of another variable**.
- Like regular variables, pointers in C++ have data types.
- A pointer should have the **same data type** as that of the variable it points to.

- **Note:** The reason we declare data types of pointers is to know how many bytes of data is used by the variable it stores the address of. If we increment (or decrement) a pointer, we increase (or decrease) the pointer by the size of the data type it points to.

Overview


int Var = 10;
int *ptr;
ptr = &Var;
ptr
0x7fffa0757dd4
0x7fff98b499e8
Address of pointer variable ptr
Var
10
Value of variable var (*ptr)
0x7fffa0757dd4
Address of variable var (Stored at ptr)

Reference Operator and Dereference Operator

Reference Operator (&)

- The *reference operator* (&) returns the address of any variable (including pointers). **For example:**

```
float a = 23.4;

// using the reference operator
cout << &a;
```

- **Output:**

```
0x7ffe0734e9b4
```

- **Note:** that the address of each variable is different in different systems.

Reference Operator and Dereference Operator

Reference Operator (&)

- As we know, pointers in C++ are used to store addresses of variables. In order to store the address of any variable in a pointer, we use the reference operator. In other words, we can assign addresses to pointers using the reference operator.
- **Example:**

```
int var = 45;
int* ptr;

ptr = &var;

cout << "The value of &var is: " << &var << endl;
cout << "The value of ptr is: " << ptr;
```

Output:

```
The value of &var is: 0x7ffc1e98dfa4
The value of ptr is: 0x7ffc1e98dfa4
```

Reference Operator and Dereference Operator

- **Dereference Operator (*)**
  - The **Asterisk symbol (*)** is called **dereference operator** when it is used with pointers. We can access the values stored in a variable to which pointer points to, by using the pointer's identifier and the dereference operator.
  - In relation to pointers, the asterisk symbol (*) has two different meanings.
    - When * is used in a variable declaration, the value written on the right-hand side of the = sign should be an address of a variable (present in the memory).
    - The unary operator (*) when used with a pointer allows us to retrieve or assign a value stored in the memory location pointed by the pointer. The unary operator can be read as "value pointed by".

Reference Operator and Dereference Operator

Dereference Operator (*) - Example

```
int var = 45;
int* ptr;

ptr = &var;

// using the dereference operator
cout << "The value returned by ptr is: " << ptr << endl;
cout << "The value returned by *ptr is: " << *ptr;
```

Output:

```
The value returned by ptr is: 0x7fff40bf6674
The value returned by *ptr is: 45
```

Pointer Declaration Syntax

- Pointers in C++ are declared using the following syntax:

```
datatype *pointer_name;
// or
datatype* pointer_name;
// or
datatype * pointer_name;
```

- We use the asterisk (*) symbol to designate a variable as a pointer in C++. The **asterisk symbol can be placed anywhere** before the pointer name and after the datatype.
- If we have to declare two (or more) pointers together in the same line, we will need to **use the asterisk symbol before each variable name**. For example:

```
int* var1, *var2; // Both var1 and var2 are pointers


int* var1, var2; // var1 is a pointer, var2 is an integer variable
```

How to use Pointers in C++?

- We have to follow a few steps to use pointers in C++:
  1. Define a pointer variable
  2. Assigning the address of a variable to a pointer using the unary operator (&) which returns the address of that variable.
  3. Accessing the value stored in the address using unary operator (*) which returns the value of the variable located at the address specified by its operand.
- The following table shows the symbols that are used with pointers.

Symbol	Name	Description
&	Address of operator	Used to find the address of a variable
*	Indirection operator	Used to access the value at an address

How to use Pointers in C++?

```cpp
#include <iostream>
using namespace std;

int main()
{
    int var = 23;

    int *ptr;

    ptr = &var;

    cout << "Initial value of var is: " << var << endl;
    cout << "Initial value of *ptr is: " << *ptr << endl << endl;

    // changing the value of var using ptr
    *ptr = 50;

    cout << "New value of *ptr is: " << *ptr << endl;
    cout << "New value of var is: " << var << endl;

    return 0;
}
```

Output:

```
Initial value of var is: 23
Initial value of *ptr is: 23
New value of *ptr is: 50
New value of var is: 50
```

Ways to Pass C++ Arguments to a Function

- In C++, we can pass arguments to functions in three different ways. These are:
  - Call By Value
  - Call By Reference with Pointer Argument
  - Call By Reference with Reference Argument

Ways to Pass C++ Arguments to a Function

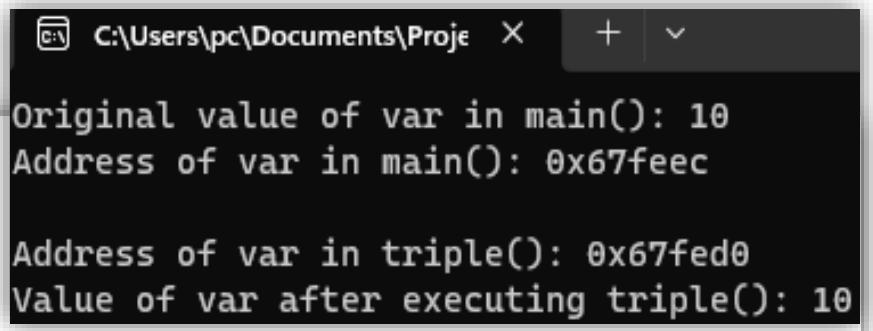
- Call By Value
  - By default, C++ uses the call by value method.
  - This method copies the real value of an argument into the function's parameter.
  - So, if the parameter inside the function is changed, it will not affect the argument.

  - From the below example, we can observe that the address of variable **var** was different inside the function **triple()**. Also, altering **var** inside the function **triple()** did not have any impact on **var** present in the **main()** function.

Ways to Pass C++ Arguments to a Function

Call By Value- Example

```
#include <iostream>
using namespace std;

// Call by Value
int triple(int var){
    // address of var in triple() is different from var in main()
    cout << "Address of var in triple(): " << &var << endl;
    var = var * 3;
    return var;
}

int main(){
    int var = 10;
    cout << "Original value of var in main(): " << var << endl;
    cout << "Address of var in main(): " << &var<< endl << endl;

    triple(var);

    cout << "Value of var after executing triple(): " << var;
    return 0;
}
```



Ways to Pass C++ Arguments to a Function

Call By Reference With Pointer Argument

- In call by reference with pointer argument, we pass the variables' address as the arguments to the parameters of a function.
- That's why the original variables are modified if we make changes to the function's parameters.

- Because we used call by reference in the following example, the address of variable **var** was the same in both **triple()** and **main()** functions. This means both **var** variables share the same memory location. That's why changing the value of **var** inside **triple()**, resulted in the change in **var** inside **main()**.

Ways to Pass C++ Arguments to a Function

Call By Reference With Pointer Argument- Example

```
#include <iostream>
using namespace std;

// Call by Reference with pointer argument
void triple(int *var){
    // Note that var is a pointer here
    // address of var in triple() is same as var in main()
    cout << "Address of var in triple(): " << var << endl;
    *var = *var * 3;
}


int main(){
    int var = 10;
    cout << "Original value of var in main(): "<<var<< endl;
    cout << "Address of var in main(): " << &var << endl;

    // passing the address of var in triple()
    triple(&var);
    cout << "Value of var after executing triple(): " << var;
    return 0;
}
```

```
Original value of var in main(): 10
Address of var in main(): 0x67feec

Address of var in triple(): 0x67feec
Value of var after executing triple(): 30
```

Ways to Pass C++ Arguments to a Function

Call By Reference With Reference Argument

- In call by reference with reference argument, we pass the variables' address as the arguments.
- There is only one difference between the two types of call by references.
  - **Call by reference with pointer argument** takes pointers (that point towards the memory location of arguments) as the function's parameters.
  - **Call by reference with reference argument** takes the original variable itself (not a copy of variables) as the function's parameters.
- To pass the original variable as a function's parameter, we use the reference operator (&) in the declaration of a function's parameters.

Ways to Pass C++ Arguments to a Function

Call By Reference With Reference Argument- Example

```
#include <iostream>
using namespace std;

// Call by Reference with reference argument
void triple(int &var){
    // Note that var is an integer variable, not a pointer
    // address of var in triple() is same as var in main()
    cout << "Address of var in triple(): " << &var << endl;
    // no need of dereference operator
    var = var * 3;
}

int main()
{
    int var = 10;
    cout << "Original value of var in main(): " << var << endl;
    cout << "Address of var in main(): " << &var
         << endl << endl;

    // passing var in triple()
    triple(var);
    cout << "Value of var after executing triple(): " << var;
    return 0;
}
```

```
Original value of var in main(): 10
Address of var in main(): 0x67feec

Address of var in triple(): 0x67feec
Value of var after executing triple(): 30
```

Void Pointers

- A **void pointer** in C++ is a special pointer that can point to objects of any data type.
- In other words, a void pointer is a general purpose pointer that can store the address of any data type, and it can be typecasted to any type.
- A void pointer is not associated with any particular data type.

- The size of a void pointer is different in different systems.
  - In 16-bit systems, the size of a void pointer is 2 bytes.
  - In a 32-bit system, the size of a void pointer is 4 bytes.
  - In a 64-bit system, the size of a void pointer is 8 bytes.

Void Pointers

- In C++, we use the **void** keyword to declare a void pointer.
- It is used with a * symbol after the **void keyword**.
- **Syntax:** void* pointer_name;

- In the following example, we used a void pointer to store the address of an integer variable and then used the same pointer to store the address of a character variable.

```
int int_var = 63;
char char_var = 'f';

void *ptr = &int_var;

ptr = &char_var;
```

Void Pointers

- In the following example, we declared a void pointer variable and a **float** variable. Then, we stored the address of the **float** variable in the void pointer variable.

```cpp
#include <iostream>
using namespace std;

int main()
{
    void* ptr;

    float marks = 85.25;

    // initializing void pointer variable
    ptr = &marks;

    // printing the address of the marks variable
    cout << "The value of &marks is: " << &marks << endl;

    cout << "The value of ptr is: " << ptr <<endl;

    return 0;
}
```

```
C:\Users\pc\Desktop\Pointer
The value of &marks is: 0x67fef8
The value of ptr is: 0x67fef8

Process returned 0   execution time : 0.087 s
Press any key to continue.
```

Invalid Pointers

- A pointer in C++ is considered valid if:
  - It is a NULL pointer value, or
  - It points to an object, or
  - It does not points to an out of bounds element of an array that is, other than range array_name to arrry_name + array_size (both inclusive).
- A pointer that satisfies no condition of three conditions written above, is known as an invalid pointer. A valid pointer may become invalid if the object to which (or after which) it points ends its life cycle i.e., the memory location to which it points to gets deallocated.
- Invalid pointers may or may not raise errors in a program. Accessing these pointers can lead to unexpected behavior of a program. Hence, we should always avoid invalid pointers.

Invalid Pointers

- In the following example, we created two pointers ptr1 and ptr2. The pointer ptr1 is invalid because it does not point to any address. The pointer ptr2 is invalid because &arr[0] + 7 does not point to any object in this program.

```
int main(){
    /*invalid pointer because
    it does not point to anything */
    int *ptr1;

    int arr[5];

    /* invalid pointer because
    it points to a non-existing address*/
    int *ptr2 = &arr[0] + 7;

    return 0;
}
```

NULL Pointers

- We can assign NULL to a pointer in C++. The value of NULL is zero. A pointer that is assigned a NULL value is called a null pointer.

- NULL allows us to create valid pointers, without storing the address of any variable in the pointer.

- It is recommended to assign NULL during the pointer declaration. Otherwise, the compiler may generate a run time error.

NULL Pointers

- As we can observe, we created a null pointer and then printed its value in the below example.

```cpp
#include <iostream>
using namespace std;

int main ()
{
    // defining a null pointer
    int *ptr = NULL;

    cout << "The value of ptr: " << ptr;

    return 0;
}
```

Output:

```
The value of ptr: 0
```

Arrays of pointers

- An array of pointers is an array that consists of variables of pointer type, which means that the variable is a pointer addressing to some other element.
- Suppose we create an array of pointer holding 5 integer pointers; then its declaration would look like:

```
int *ptr[5]; // array of 5 integer pointer.
```

- The element of an array of a pointer can also be initialized by assigning the address of some other element.

```
int a; // Variable declaration.
ptr[2] = &a;
```

- We can also retrieve the value of 'a' be dereferencing the pointer.

```
*ptr[2];
```

Arrays of pointers

```
#include <iostream>
using namespace std;
const int MAX = 3;

int main () {
   int  var[MAX] = {10, 100, 200};
   int *ptr[MAX];

   for (int i = 0; i < MAX; i++) {
      ptr[i] = &var[i]; //assign the address of integer.
   }

   for (int i = 0; i < MAX; i++) {
      cout << "Value of var[" << i << "] = ";
      cout << *ptr[i] << endl;
   }
   return 0;
}
```

```
C:\Users\pc\Documents\Proje
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200

Process returned 0   execution time : 0.101 s
Press any key to continue.
```

Arrays and Pointers

- An array name contains the address of the first element of the array which acts like a constant pointer.
- It means, the address stored in the array name can't be changed.
- The name of an array acts like a pointer because the address of the first element of an array is stored in its name. So, if a pointer contains the address of the first element of an array, we can use that pointer to access all the elements of the array.
- Not only can a pointer store the address of a single variable, it can also store the address of cells of an array.

Arrays and Pointers

- Consider this example:

```
int *ptr;
int arr[5];

// store the address of the first
// element of arr in ptr
ptr = arr;
```

- Here, ptr is a pointer variable while arr is an int array. The code ptr = arr; stores the address of the first element of the array in variable ptr.
- Notice that we have used arr instead of &arr[0]. This is because both are the same. So, the code below is the same as the code above.

```
int *ptr;
int arr[5];
ptr = &arr[0];
```

- The addresses for the rest of the array elements are given by &arr[1], &arr[2], &arr[3], and &arr[4].

Arrays and Pointers

- Point to Every Array Elements
  - Suppose we need to point to the fourth element of the array using the same pointer ptr.
  - Here, if ptr points to the first element in the above example then ptr + 3 will point to the fourth element.
  - **For example,**

```
int *ptr;
int arr[5];
ptr = arr;

ptr + 1 is equivalent to &arr[1];
ptr + 2 is equivalent to &arr[2];
ptr + 3 is equivalent to &arr[3];
ptr + 4 is equivalent to &arr[4];
```

Arrays and Pointers

- Point to Every Array Elements
  - Similarly, we can access the elements using the single pointer.
  - **For example,**

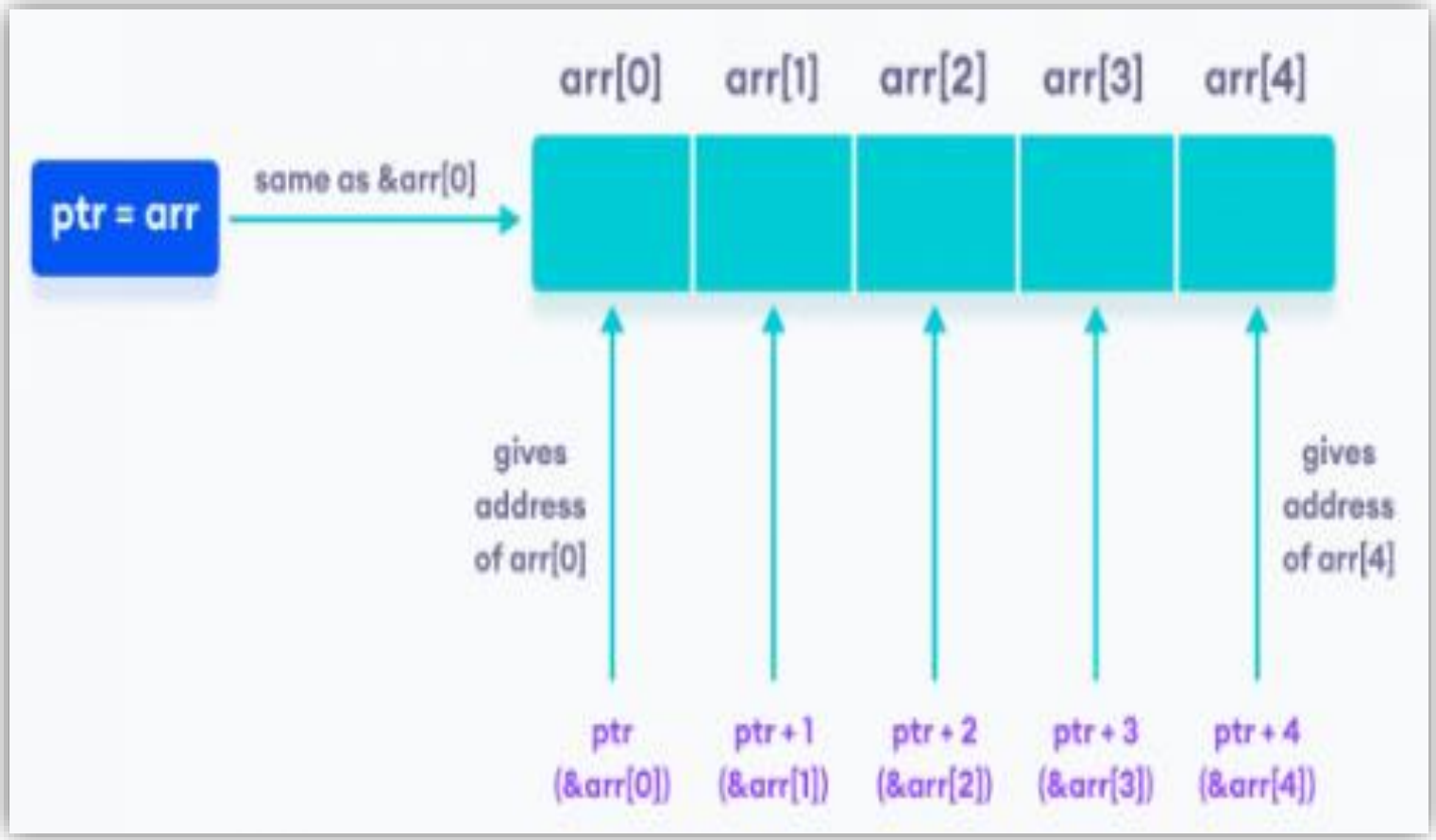
```
// use dereference operator
*ptr == arr[0];
*(ptr + 1) is equivalent to arr[1];
*(ptr + 2) is equivalent to arr[2];
*(ptr + 3) is equivalent to arr[3];
*(ptr + 4) is equivalent to arr[4];
```

- Suppose if we have initialized ptr = &arr[2]; then

```
ptr - 2 is equivalent to &arr[0];
ptr - 1 is equivalent to &arr[1];
ptr + 1 is equivalent to &arr[3];
ptr + 2 is equivalent to &arr[4];
```

Arrays and Pointers

- Point to Every Array Elements



Point to Every Array Elements

- **Note:** The address between ptr and ptr + 1 differs by 4 bytes. It is because ptr is a pointer to an int data. And, the size of int is 4 bytes in a 64-bit operating system.

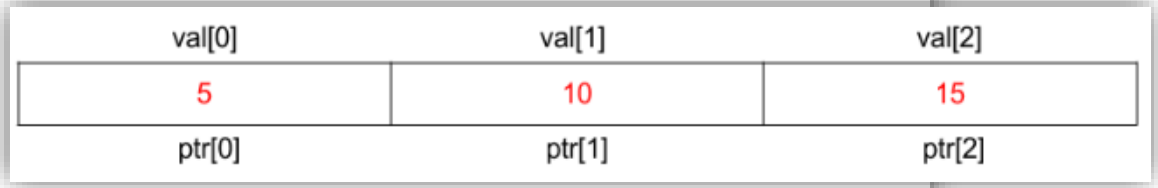
- Similarly, if pointer ptr is pointing to char type data, then the address between ptr and ptr + 1 is 1 byte. It is because the size of a character is 1 byte.

Arrays and Pointers

- For example, if we have an array named val then **val** and **&val[0]** can be used interchangeably.

```
// C++ program to illustrate Array Name as Pointers
#include <iostream>
using namespace std;
void geeks()
{
    // Declare an array
    int val[3] = { 5, 10, 20 };

    // declare pointer variable
    int* ptr;

    // Assign the address of val[0] to ptr
    // We can use ptr=&val[0];(both are same)
    ptr = val;
    cout << "Elements of the array are: ";
    cout << ptr[0] << " " << ptr[1] << " " << ptr[2];
}
// Driver program
int main() {
    geeks();
    return 0;
}
```

```
C:\Users\pc\Desktop\Pointer
Elements of the array are: 5 10 20
Process returned 0   execution time : 0.249 s
Press any key to continue.
```



Pointer Expressions and Pointer Arithmetic

- A limited set of arithmetic operations can be performed on pointers which are:
  - incremented (++)
  - decremented (--)
  - an integer may be added to a pointer (+ or +=)
  - an integer may be subtracted from a pointer (- or -=)
  - difference between two pointers (p1-p2)
- **Note**: Pointer arithmetic is meaningless unless performed on an array.

Pointer Expressions and Pointer Arithmetic

Example 1: Using Increment Operator

- When we increment a pointer using the increment operator (++), the address of the pointer increases. The increase in address of the pointer is equal to the size of its data type.
- As all the elements of the array are stored in contiguous memory, we can use the increment operator on pointers to access the elements of an array.
- In the example, we used **ptr++** to access each element of the array *arr*. Since **ptr** had an **int** type, the address was increased by 44 (because size of an **int** is 44) when we used **ptr++**.

Pointer Expressions and Pointer Arithmetic

- Example 1: Using Increment Operator

```cpp
#include <iostream>
using namespace std;

int main () {
    int  arr[3] = {10, 20, 30};

    // storing address of arr in a pointer
    int  *ptr = arr;

    for (int i = 0; i < 3; i++)
    {
        cout << "Value of var[" << i << "] is: "
             << *ptr << endl;
        cout << "Address of var[" << i << "] is: "
             << ptr << endl << endl;

        // point to the next location
        ptr++;
    }
    return 0;
}
```

Output:

```
Value of var[0] is: 10
Address of var[0] is: 0x7fff9e39b200

Value of var[1] is: 20
Address of var[1] is: 0x7fff9e39b204

Value of var[2] is: 30
Address of var[2] is: 0x7fff9e39b208
```

Pointer Expressions and Pointer Arithmetic

Example 2: Using Decrement Operator

- The decrement operator (--) is similar to the increment operator. Decrement operator decreases the address of a pointer by the size of its data type.
- The decrement operator can also be used with arrays to access their elements.

- In the following example, the pointer ptr was pointing to the last element of the array arr. In order to access each element of the array using ptr, we used ptr-- inside the for loop.

Pointer Expressions and Pointer Arithmetic

- Example 2: Using Decrement Operator

```cpp
#include <iostream>
using namespace std;

int main () {
    int  arr[3] = {10, 20, 30};

    // storing address of last element of arr in a pointer
    int  *ptr = &arr[2];

    for (int i = 2; i >= 0; i--)
    {
        cout << "Value of var[" << i << "] is: "
             << *ptr << endl;
        cout << "Address of var[" << i << "] is: "
             << ptr << endl << endl;

        // point to the previous location
        ptr--;
    }
    return 0;
}
```

Output:

```
Value of var[2] is: 30
Address of var[2] is: 0x7ffd19a65bb4

Value of var[1] is: 20
Address of var[1] is: 0x7ffd19a65bb0

Value of var[0] is: 10
Address of var[0] is: 0x7ffd19a65bac
```

Pointer Expressions and Pointer Arithmetic

Example 3: Addition and Subtraction

- If we add 3 to a pointer (ptr + 3), the pointer will point to the memory address located 3 places ahead of the current address. In other words, the pointer will point to an address that is three times the size of the pointer's data type (3 * size_of_pointer_type).
- The subtraction operation is similar to addition. In case of the subtraction operation in pointers, if we subtract 1 from the pointer (ptr - 1), the pointer will point to the previous memory address.
- In the following example, **ptr1 + 2** is equivalent to **&arr[2]**, and **ptr2 - 1** is equivalent to **&arr[3]**.

Pointer Expressions and Pointer Arithmetic

- Example 3: Addition and Subtraction

```
int main () {
    int  arr[5] = {10, 20, 30, 40, 50};

    int *ptr1, *ptr2;

    // assigning address of 1nd element of arr to ptr1
    ptr1 = arr;

    // assigning address of 5th element of arr to ptr2
    ptr2 = &arr[4];

    cout << "Value of ptr1 is: " << ptr1 << endl;

    // using addition
    cout << "Value of ptr1 + 2 is: " << ptr1 + 2 << endl
         << endl;

    cout << "Value of ptr2 is: " << ptr2 << endl;

    // using subtraction
    cout << "Value of ptr2 - 1 is: " << ptr2 - 1 << endl << endl;

    return 0;
```

Output:

```
Value of ptr1 is: 0x7ffeec420110
Value of ptr1 + 2 is: 0x7ffeec420118

Value of ptr2 is: 0x7ffeec420120
Value of ptr2 - 1 is: 0x7ffeec42011c
```

Pointer Expressions and Pointer Arithmetic

- Example 4: Subtracting two Pointers
  - Subtracting two pointers is a valid operation although adding two pointers is not a valid operation.
  - The subtraction of two pointers is possible only when they have the same data type.
  - The subtraction of two pointers returns the increments between the two pointers.

Pointer Expressions and Pointer Arithmetic

- Example 4: Subtracting two Pointers

```
#include <iostream>
using namespace std;

int main (){
  int Var = 10;
  int *p1, *p2;

  p1 = &Var;
  p2 = p1 + 5;

  cout<<"(p2 - p1) : "<<(p2-p1);
  return 0;
}
```

```
C:\Users\pc\Documents\Proje
(p2 - p1) : 5
Process returned 0   execution time : 0.081 s
Press any key to continue.
```

Pointer Expressions and Pointer Arithmetic

- Example 5: Pointer Comparison
  - Pointers can be compared using relational operators, such as ==, <, >, <= and >=. If two pointers are related to each other such as pointing to the elements of same array, then they can be compared meaningfully.

  - In the example below, pointer comparison property is used to iterate over an array. A pointer pointing to an element of an array is incremented in each loop until its value becomes greater than the address of the last element of the array.

Pointer Expressions and Pointer Arithmetic

- Example 5: Pointer Comparison

```
#include <iostream>
using namespace std;

int main (){
  int Arr[5] = {10, 20, 30, 40, 50};
  int *p1;

  p1 = Arr;    //pointing to Arr[0]

  while(p1 <= &Arr[4]) {
    cout<<"p1 : "<<p1<<"\n";
    cout<<"*p1 : "<<*p1<<"\n";
    p1++;
  }
  return 0;
}
```

```
C:\Users\pc\Documents\Proje
p1 : 0x67fed8
*p1 : 10
p1 : 0x67fedc
*p1 : 20
p1 : 0x67fee0
*p1 : 30
p1 : 0x67fee4
*p1 : 40
p1 : 0x67fee8
*p1 : 50
```

Pointer Expressions and Pointer Arithmetic

- *Programming Exercises:*
  1. Write a program that initializes integer values to an array and displays the value of the array on the screen using pointer notation and pointer arithmetic in C++.
  2. Write a program that inputs values into an array and displays the odd value of array on screen using pointer notation and pointer arithmetic in C++.
  3. Write a program that inputs value into array and display in reverse order on screen using pointer notation and pointer arithmetic in C++.

Pointer and string

- Strings can also be declared using pointers.

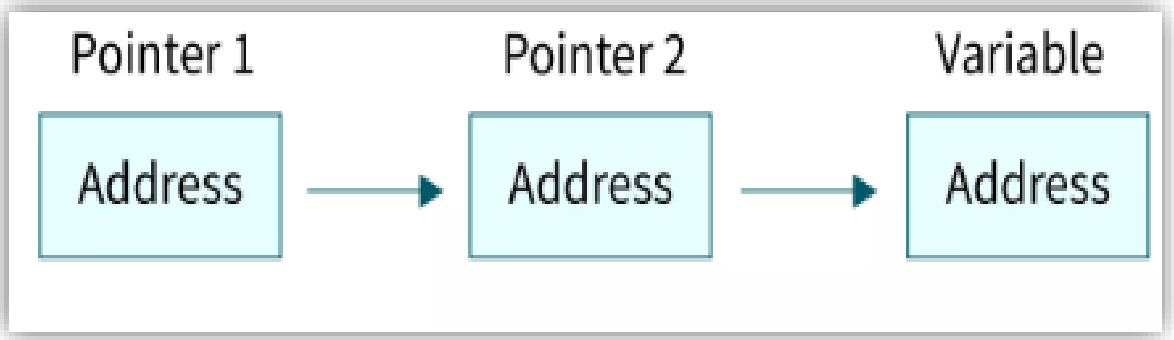
```
int main(){
    char name[ ]= "Computer";
    char *p;
    /*for string, only this declaration
      will store its base address */
    p = name;
    while( *p != '\0'){
        cout << *p;
        p++;
    }
    return 0;
}
```

```
C:\Users\pc\Documents\Proje
Computer
Process returned 0   execution time : 0.071 s
Press any key to continue.
```

- In the above example, since **p** stores the address of name[0], therefore the value of ***p** equals the value of name[0] i.e., 'S'. So in while loop, the first character gets printed and **p++** increases the value of **p** by 1 so that now p+1 points to name[1]. This continues until the pointer reaches the end of the string i.e., before *p becomes equal to '\0'.

Pointers to Pointers

- A pointer to a pointer is a chain of pointers. When we define a pointer to a pointer, the first pointer points to the second pointer, and the second pointer points to the actual variable.
- To declare a pointer to a pointer, we use one unary operator (*) for each level of chaining of pointers.



- In the following example, we created a variable var and two pointers ptr1 and ptr2. The address of var was stored in ptr1, while the address of ptr1 was stored in ptr2.

Pointers to Pointers

```cpp
#include <iostream>
using namespace std;

int main(){
    float var = 10;
    cout << "Value of &var is: " << &var << endl << endl;

    // declaring a pointer
    float *ptr1;

    // declaring a pointer to a pointer
    float **ptr2;

    // assigning address of variable var to ptr1
    ptr1 = &var;

    cout << "Value of ptr1 is: " << ptr1 << endl;
    cout << "Value of &ptr1 is: " << &ptr1 << endl << endl;

    // assigning address of pointer ptr1 to ptr2;
    ptr2 = &ptr1;

    cout << "Value of ptr2 is: " << ptr2 << endl;
    cout << "Value of *ptr2 is: " << *ptr2 << endl << endl;

    return 0;
}
```

```
Value of &var is: 0x67fee8

Value of ptr1 is: 0x67fee8
Value of &ptr1 is: 0x67fee4

Value of ptr2 is: 0x67fee4
Value of *ptr2 is: 0x67fee8
```

Common Mistakes When Working With Pointers

- Let us now take a look at a few common mistakes people make while using pointers.

```
int *ptr, var;


// Wrong
ptr = var;


// Correct
ptr = &var;
```

- In the above example, var is a variable, not an address. So, we need to write &var to store the address of var in ptr.

Common Mistakes When Working With Pointers

- In the above example, ***ptr** denotes the value stored in the variable **var**, while **&var** denotes the address of **var**. If we want to store the value of **var** in ***ptr**, we need to remove **&** from **&var**.

```
int var = 10, *ptr;

ptr = &var;

var = 20;

// Wrong
*ptr = &var;

// Correct
*ptr = var;
```

Advantages of Using Pointers

- Following are the advantages of using pointers:
  - We can dynamically allocate and de-allocate memory using pointers.
  - Pointers are more efficient in handling arrays and data tables.
  - Pointers can be used to return multiple values from a function. This can be done by passing the arguments with their addresses and making changes to the argument's values using pointers.
  - Pointers are efficient in handling dynamic data structures like linked lists, trees, etc.

Conclusion

- Pointers can store the memory address of variables, other pointers, and functions.
- We can pass arguments to a function in three ways: call by value, call by reference with pointer argument, and call by reference with reference argument.
- We can perform four arithmetic operations on arrays: increment, decrement, addition, and subtraction.
- We can use the const keyword with pointers and we can iterate through the elements of an array using pointers.
- A pointer may become invalid if it is not a NULL pointer, not pointing to any object/memory, or pointing to an array index which is out of bounds.
- Pointers without a data type are called void pointers. Additionally, we can assign NULL to pointers.

Chapter Six

C++ Structures

Outline

- Introduction
- Defining the structure
- Accessing Structure Members
- Single Structure
- Arrays of Structure
- Structures as Function Arguments
- Pointers to Structs

User Defined Data Types

- **Structure**
  - **Member of the structure**
    - **Variables**
    - **Functions**

- **Class**
  - **Member of the class**
    - **Variables**
    - **Functions**

Introduction

- **C++ struct**, short for C++ Structure, is an user-defined data type available in C++. It allows a user to combine data items of (possibly) different data types under a single name.

- C++ structs are different from arrays because arrays *only* hold data of similar data-types; C++ struct, on the other hand, can store data of multiple data-types.

- Each element in the structure is called a ***member***.

How to create a structure?

- **Syntax:**

```
struct structureName{
    member1;
    member2;
    member3;
    .
    .
    .
    memberN;
};
```

- C++ Structures can contains two types of members:
  - **Data Member:** These members are normal C++ variables. We can create a structure with variables of different data types in C++.
  - **Member Functions:** These members are normal C++ functions. Along with variables, we can also include functions inside a structure declaration.

How to create a structure?

- **Example:**

```
struct Student{
    // Data Members
    int roll;
    int age;
    int marks;

    // Member Functions
    void printDetails()
    {
        cout<<"Roll = "<<roll<<"\n";
        cout<<"Age = "<<age<<"\n";
        cout<<"Marks = "<<marks;
    }
}
```

- In the above structure, the data members are:
  - three integer variables to store *roll number, age and marks* of any student and
  - the member function is *printDetails()* which is printing all of the above details of any student.

How to declare structure variables?

- A structure variable can either be declared with structure declaration or as a separate declaration like basic types.

```
1  #include <iostream>
2  using namespace std;
3
4  // A variable declaration with structure declaration.
5  struct Point{
6  int x, y;
7  } p1; // The variable p1 is declared with 'Point'
8
9  // A variable declaration like basic data types
10 struct Point{
11 int x, y;
12 };
13
14 int main(){
15     // The variable p1 is declared like a normal variable
16     struct Point p1;
17 }
```

Access Structure Members

- To access members of a structure, use the dot syntax (.):
- **Example**: Assign data to members of a structure and print it:

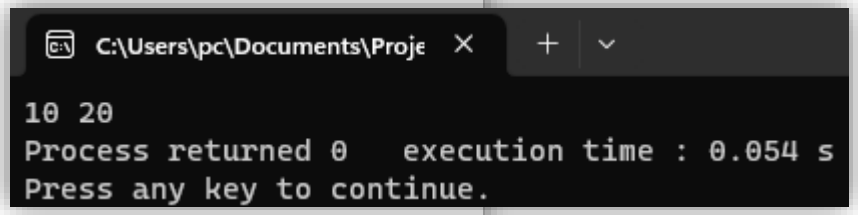
```cpp
#include <iostream>
using namespace std;

// Create a structure variable called myStructure
struct {
  int myNum;
  string myString;
} myStructure;

int main(){
    // Assign values to members of myStructure
    myStructure.myNum = 1;
    myStructure.myString = "Hello World!";

    // Print members of myStructure
    cout << myStructure.myNum << "\n";
    cout << myStructure.myString << "\n";
}
```

```
C:\Users\pc\Documents\Proje
1
Hello World!

Process returned 0   execution time : 0.057 s
Press any key to continue.
```

An array of structures

- Like other primitive data types, we can create an array of structures.

```cpp
#include <iostream>
using namespace std;

struct Point {
    int x, y;
};

int main(){
    // Create an array of structures
    struct Point arr[10];

    // Access array members
    arr[0].x = 10;
    arr[0].y = 20;

    cout << arr[0].x << " " << arr[0].y;
    return 0;
}
```



Structures as function arguments

- In C++, we can pass a structure variable to a function as argument like we pass any other variable to a function.
- Structure variable is passed using call by value.
- To take a structure variable as argument, function must declare a structure argument in it's declaration.
- Any change in the value of formal parameter inside function body, will not affect the value of actual parameter.

Structures as function arguments

- Example:

```
struct employee {
    char name[100];
    int age;
    float salary;
    char department[50];
};

void printEmployeeDetails(struct employee emp);
```

- We can also pass address of a structure to a function. In this case, any change in the formal parameter inside function's body will reflect in actual parameter also. To take a structure pointer as parameter a function declare a structure pointer as it's formal parameter.

```
void printEmployeeDetails(struct employee emp);
```

- Like any other inbuilt data type, we can also pass individual member of a structure as function argument.

Structures as function arguments

Example:

```
1  #include <iostream>
2  using namespace std;
3
4  struct employee {
5   char name[100];
6   int age;
7   float salary;
8   char department[50];
9  };
10 // This function takes structure variable as parameter
11 void printEmployeeDetails(struct employee emp){
12    cout << "\n*** Employee Details ***\n";
13    cout << "Name : " << emp.name << "\nAge : "<< emp.age << "\nSalary :
14         << emp.salary << "\nDepartment : "<< emp.department;
15 }
16 // This function takes structure pointer as parameter
17 void printEmployeeDetails(struct employee *emp){
18    cout << "\n--- Employee Details ---\n";
19    cout << "Name : " << emp->name << "\nAge : "<< emp->age << "\nSalary
20         << emp->salary << "\nDepartment : "<< emp->department;
21 }
22 void printAge(int age){
23     cout << "\n\nAge = " << age;
24 }
```

Structures as function arguments

Example:

```
27  int main(){
28      struct employee manager, *ptr;
29
30      printf("Enter Name, Age, Salary and Department of Employee\n");
31      // Assigning data to members of structure variable
32      cin >> manager.name;
33      cin >> manager.age;
34      cin >> manager.salary;
35      cin >> manager.department;
36
37      // Passing structure variable to function
38      printEmployeeDetails(manager);
39
40      // Passing address of structure variable to a function
41      printEmployeeDetails(&manager);
42      /* Passing structure member to function */
43      printAge(manager.age);
44
45      return 0;
46  }
```

Structures as function arguments

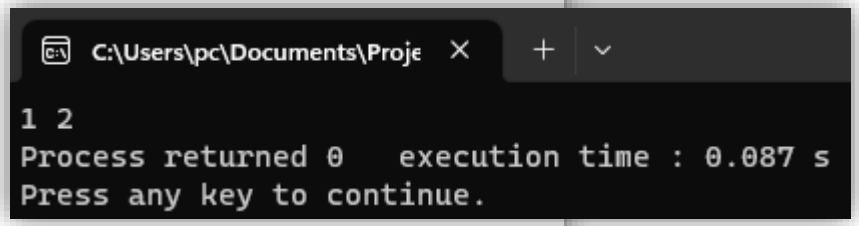
Example: *Output*

```
Enter Name, Age, Salary and Department of Employee
Nolawi
29
12879
CS

*** Employee Details ***
Name : Nolawi
Age : 29
Salary : 12879
Department : CS
--- Employee Details ---
Name : Nolawi
Age : 29
Salary : 12879
Department : CS

Age = 29
```

A structure pointer

- Like primitive types, we can have pointer to a structure. If we have a pointer to structure, members are accessed using arrow (->) operator instead of the dot (.) operator.

```cpp
#include <iostream>
using namespace std;

struct Point {
    int x, y;
};

int main(){
    struct Point p1 = { 1, 2 };

    // p2 is a pointer to structure p1
    struct Point* p2 = &p1;

    // Accessing structure members using
    // structure pointer
    cout << p2->x << " " << p2->y;
    return 0;
}
```



Chapter Seven

File Management in C++

What is File Handling in C++?

- File handling in C++ is a mechanism to store the output of a program in a file and help perform various operations on it. Files help store these data permanently on a storage device.
- The term “Data” is commonly referred to as known facts or information. In the present era, data plays a vital role. It helps to describe, diagnose, predict or prescribe. But to achieve all this, we need to store it somewhere. You all would argue that there are so many text editors like ‘Notepad’ and ‘MS Office’, which help us store data in the form of text. You are right! But here we are discussing at a level of programming. In contrast, text editors like ‘Notepad’ and ‘MS Office’ are pre-built and cannot be accessed at the programming level to store data.

What is File Handling in C++?

- Almost every programming language has a 'File Handling' method to deal with the storage of data. In this article, we will learn about file handling in C++. But, before that, if you are a newbie at C++, you could check out this free course on C++ to learn the basics.
- Now, This topic of file handling is further divided into sub-topics:
  - Create a file
  - Open a file
  - Read from a file
  - Write to a file
  - Close a file

fstream library

- Before diving into each sub-topics, let us first learn about the header file we will be using to gain access to the file handling method.
- In C++, fstream library is used to handle files, and it is dealt with the help of three classes:
  - **ifstream**: This class helps create and write the data to the file obtained from the program's output. It is also known as the input stream.
  - **ifstream**: We use this class to read data from files and also known as the input stream.
  - **fstream**: This class is the combination of both ofstream and ifstream. It provides the capability of creating, writing and reading a file.
- To access the following classes, you must include the fstream as a header file like how we declare iostream in the header.

File Operations in C++

- C++ provides us with four different operations for file handling. They are:
  - **open()** – This is used to create a file.
  - **read()** – This is used to read the data from the file.
  - **write()** – This is used to write new data to file.
  - **close()** – This is used to close the file.

Opening files in C++

- To read or enter data to a file, we need to open it first. This can be performed with the help of 'ifstream' for reading and 'fstream' or 'ofstream' for writing or appending to the file. All these three objects have open() function pre-built in them.
- Syntax: open(FileName , Mode);
  - FileName – It denotes the name of file which has to be opened.
  - Mode – There different mode to open a file and it explained in the next slide.

Opening files in C++

- **File opening modes:**
  1. **ios::in** - Opens an existing file for input
  2. **ios::out** - Opens a file for output. This flag implies ios::trunc if it is not combined with one of the flags ios::in or ios::app or ios::ate.
  3. **ios::app** - Opens a file for output at the end-of-file.
  4. **ios::trunc** - An existing file is truncated to zero length.
  5. **ios::ate** - Open and seek to end immediately after opening. Without this flag, the starting position after opening is always at the beginning of the file.
  6. **ios::binary** - Perform input and output in binary mode.
  7. **ios::noreplace –** open fails if the file already exist.
  8. **ios::nocreate** - open fails if the file does not exist.

Opening files in C++

- **Default settings when opening a file**
  - The constructor and the method open() of all stream classes use the following default values:

Class	***Fags***
• **ifstream**	**ios::in**
• **ofstream**	**ios::out | ios::trunc**
• **fstream**	**ios::in | ios::out**

- In C++, we can use two modes simultaneously with the help of | (OR) operator.

Opening files in C++

- **Example:**

```
1  #include<iostream>
2  #include<fstream>
3  using namespace std;
4  int main(){
5      fstream FileOpeningObject;
6      FileOpeningObject.open("FileName", ios::out);
7      if (!FileOpeningObject){
8          cout<<"Error while creating the file";
9      }
10     else{
11         cout<<"File created successfully";
12         FileOpeningObject.close();
13     }
14     return 0;
15 }
```

Writing to file in C++

- Till now, we learned how to create the file using C++. Now, we will learn how to write data to file which we created before. We will use fstream or ofstream object to write data into the file and to do so; we will use stream insertion operator (<<) along with the text enclosed within the double-quotes.

- With the help of open() function, we will create a new file named 'FileName' and then we will set the mode to 'ios::out' as we have to write the data to file.
- **Syntax**: FileOpeningObject<<"Insert the text here";

Writing to file in C++

- **Example**:

```
1  #include<iostream>
2  #include<fstream>
3  using namespace std;
4  int main() {
5      fstream FileName;
6      FileName.open("FileName.txt", ios::out);
7      if (!FileName) {
8          cout<<" Error while creating the file ";
9      }
10     else {
11         cout<<"File created and data got written to file";
12
13         FileName<<"This string will writen to the file ";
14         FileName.close();
15     }
16     return 0;
17 }
```

Reading from file in C++

- Getting the data from the file is an essential thing to perform because without getting the data, we cannot perform any task. But don't worry, C++ provides that option too. We can perform the reading of data from a file with the CIN to get data from the user, but then we use CIN to take inputs from the user's standard console. Here we will use fstream or ifstream.
- **Syntax**: FileOpeningObject<<variable;

Reading from file in C++

- **Example**:

Read Char by Char

```
int main(){//Declare and open a text file
ifstream openFile("data.txt");
 char ch;
 //do until the end of file
while( ! OpenFile.eof() )
{
OpenFile.get(ch); // get one character
cout << ch;   // display the character
}
OpenFile.close(); // close the file
return 0;}
```

Read a Line

```
int main(){//Declare and open a text file
ifstream openFile("data.txt");
string line;
while(!openFile.eof())
{//fetch line from data.txt and put it in a string
getline(openFile, line);
cout << line;
}
openFile.close(); // close the file
return 0; }
```

Closing a file in C++

- When a C++ program terminates it automatically flushes all the streams, release all the allocated
- memory and close all the opened files. But it is always a good practice that a programmer should
- close all the opened files before program termination. Following is the standard syntax for close()
- function, which is a member of fstream, ifstream, and ofstream objects.
- Syntax: FileStreamObject.close();

File Position Pointers

- When our program reads and writes files Randomly, it treats the file like a big Array. with arrays, we know we can add, print, or remove values in any order. we don't have to start at the first array element, sequentially looking at the Next one, until we get the Element we need. We can view our
- Random-access-file in the same way, accessing the data in any order. In Random Access file:
  - We Can Read from Anywhere from the file.
  - We Can Write to Anywhere in the file.
  - We can Modify our record.
  - We can Search any Record from file.
  - And we can Delete Any record from file:

File Position Pointers

- Special function to move **File-Pointer** within the File:

Function	Syntax	Description
seekg()	fileObject.seekg(long_num, origin);	We can move input pointer to a specified location for reading by using this function. fileObject is the pointer to the file that we want to access. long_num is the number of bytes in the file we want to skip. and origin is a value that tells to compiler, where to begin the skipping of bytes.
seekp()	fileObject.seekp(long_num, origin);	We can move Output pointer to a specified location by using this function. it's same like seekg() Function but in Case of writing:
tellg()	fileObject.tellg();	This function return the current position of Input pointer. this function don't need any argument.
tellg()	fileObject.tellp();	This function return the current position of Output pointer. this function don't need any argument.

File Position Pointers

- **origin** from the syntax of file pointers is the value that tells Compiler, where to begin the skipping of bytes specified by long_num . Origin can be any of the three values shown in the table below.

Function	Description
bad()	Returns True if an Error occurs while reading or writing Operation. (Example: in case we try to write to a file that is not open for writing or if the device where we try to write has no space left).
fail()	Returns true in the same cases as bad() plus in case that a format error happens. (as trying to read an integer number and an alphabetical character is received:).
eof()	Returns true if a file opened for reading has reached the end.
good()	Return False in the same cases in which calling any of the previous functions would return true

File Position Pointers

- **origin** from the syntax of file pointers is the value that tells Compiler, where to begin the skipping of bytes specified by long_num . Origin can be any of the three values shown in the table below.

Function	Syntax	Description
ios::beg	fileObject.seekg(0, ios::beg);	Go to Start: No matter how far into a file we have read, by using this Syntax, the file-pointer will back to the beginning of the file.
ios::cur	fileObject.seekg(0, ios::cur);	Stay at Current Position: Using this syntax, the filepointer will show its current position.
ios::end	fileObject.seekg(0, ios::end);	Go to End of the file: using this syntax, the file-pointer will point to end of the file.

File Position Pointers – Examples

This program illustrates some of file pointers

```
int main(){
   ifstream inf("Sample.dat");
   // If we couldn't open the input file stream for reading
   if (!inf) {
      // Print an error and exit
      cerr << "Uh oh, Sample.dat could not be opened for reading!" << endl;
      exit(1);
   }
   string strData;
   inf.seekg(5); // move to 5th character
   getline(inf, strData); // Get the rest of the line and print it
   cout << strData << endl;
   inf.seekg(8, ios::cur); // move 8 more bytes into file
   getline(inf, strData); // Get rest of the line and print it
   cout << strData << endl;
   inf.seekg(-15, ios::end); // move 15 bytes before end of file
   getline(inf, strData);     // Get rest of the line and print it
   cout << strData << endl;
  return 0;
}
```

This produces the result:

```
is line 1
line 2
his is line 4
```

File Position Pointers – Examples

```
1  // This program shows importance of tellp, tellg, seekp and seekg:
2  #include <iostream>
3  #include<conio>
4  #include <fstream>
5  using namespace stdint main(){
6      fstream st; // Creating object of fstream class
7      st.open("E:\studytonight.txt", ios::out); // Creating new file
8      if(!st){ // Checking whether file exist
9          cout << "File creation failed";
10     }
11     else{
12         cout << "New file created" << endl;
13         st << "Hello Friends"; //Writing to file
14         // Checking the file pointer position
15         cout << "File Pointer Position is " << st.tellp() << endl;
16         // Go one position back from current position
17         st.seekp(-1, ios::cur);
18         //Checking the file pointer position
19         cout << "As per tellp File Pointer Position is "
20         << st.tellp() << endl;
21         st.close(); // closing file
22     }
```

File Position Pointers – Examples

```
23      // Opening file in read mode
24      st.open("E:\studytonight.txt", ios::in);
25      if(!st){ //Checking whether file exist
26          cout << "No such file";
27      }
28      else{
29          char ch;
30          st.seekg(5, ios::beg); // Go to position 5 from begning.
31          cout << "As per tellg File Pointer Position is " << st.tellg()
32          << endl<< endl;; //Checking file pointer position
33          st.seekg(1, ios::cur); //Go to position 1 from beginning.
34          cout << "As per tellg File Pointer Position is " << st.tellg()
35          << endl; //Checking file pointer position
36          st.close(); //Closing file
37      }
38      getch();
39      return 0;
40  }
```

Error Handling in C++ Files

- In our each example, we are trying to use error checker to avoid error, and it's really good for us and for our program. C++ provides it's self some function, which can help us to reduce our error in our program

Function	Description
bad()	Returns True if an Error occurs while reading or writing Operation. (Example: in case we try to write to a file that is not open for writing or if the device where we try to write has no space left).
fail()	Returns true in the same cases as bad() plus in case that a format error happens. (as trying to read an integer number and an alphabetical character is received:).
eof()	Returns true if a file opened for reading has reached the end.
good()	Return False in the same cases in which calling any of the previous functions would return true

Other Functions in C++ Files

1. **Write():** This function is used with Output-Pointer object to write the Block of data. Write() Function uses the binary version of data instead of the text version. when write() is used, each value is stored in its binary format and each record is created to have a fixed length. The write function takes three arguments, first is Pointer to Character, the second is Address of the object, and last the size of Object.
   - **Syntax**: write((char *) &Obj, sizeof(Obj));

Other Functions in C++ Files

2. **Read():** his function is used with Input-Pointer object to read the block of data. It's same like read() Function, but in sense of reading.
   - **Syntax**: read((char *) &Obj, sizeof(Obj));
3. **Remove():** This Function is used to remove any file from record. it take one argument, which is the file Name to be delete.
   - **Syntax**: read((char *) &Obj, sizeof(Obj));
4. **Rename():** This function is used to rename any file. it take two argument, first is the Old Name of file, and the other is New name for that file.
   - **Syntax**: rename(Old_Name, New_Name);

Thank You!!!
Any ???